

MANUAL DE UNIX

Comandos y Administración del Sistema

Manual de Referencia de UNIX

MANUAL DE UNIX

Comandos y Administración

José Manuel Flomesta Abenza
Juan José Gallardo Gila
1º ASI

Tabla de contenido

Presentación de la reunión	I	Cómo crear un documento	4
		Más sugerencias acerca de las plantillas	4
C A P Í T U L O 1		C A P Í T U L O 3	
Cómo personalizar este manual	1	Cómo personalizar este manual	1
Acerca de los iconos de "imagen"	1	Acerca de los iconos de "imagen"	1
Los saltos de sección son clave	2	Los saltos de sección son clave	2
Acerca de imágenes y títulos	2	Acerca de imágenes y títulos	2
Cómo generar una tabla de contenido	3	Cómo generar una tabla de contenido	3
Cómo crear un índice	3	Cómo crear un índice	3
Modificar encabezados y pies de página	3		
Cómo ahorrar tiempo en el futuro	4	C A P Í T U L O 3	
Cómo crear un documento	4	Cómo personalizar este manual	1
Más sugerencias acerca de las plantillas	4	Acerca de los iconos de "imagen"	1
		Los saltos de sección son clave	2
C A P Í T U L O 2		Acerca de imágenes y títulos	2
Cómo personalizar este manual	1	Cómo generar una tabla de contenido	3
Acerca de los iconos de "imagen"	1	Cómo crear un índice	3
Los saltos de sección son clave	2	Cómo modificar encabezados y pies de	
Acerca de imágenes y títulos	2	página	3
Cómo generar una tabla de contenido	3	Cómo ahorrar tiempo en el futuro	4
Cómo crear un índice	3	Cómo crear un documento	4
Modificar encabezados y pies de página	3	Más sugerencias acerca de las plantillas	4
Cómo ahorrar tiempo en el futuro	4	Índice	5

Introducción

El sistema operativo UNIX ha evolucionado durante estos últimos años desde su invención como un experimento informático hasta llegar a convertirse en uno de los entornos informáticos más populares e influyentes del mundo. Este crecimiento se está acelerando cada vez más conforme más y más usuarios sucumben a la sorprendente flexibilidad, potencia y elegancia del sistema UNIX.

Estas son las características únicas del sistema UNIX que han conducido a este crecimiento:

■ *Herramientas software.* El sistema UNIX introdujo una nueva idea en la computación: los problemas pueden ser resueltos y las aplicaciones creadas mediante interconexión de unas cuantas piezas simples. Estas piezas son generalmente componentes completos diseñados para realizar una única tarea, y hacerla bien. Grandes aplicaciones pueden construirse a partir de secuencias de órdenes simples. Esta filosofía también se extiende al dominio del desarrollo, donde subrutinas empaquetadas se combinan para formar nuevos programas ejecutables. Este concepto básico de *reutilización del software* es una de las razones principales por la que el sistema UNIX resulta ser un entorno muy productivo en el que trabajar.

■ *Portabilidad.* UNIX ha sido trasvasado a casi cualquier ordenador moderado o grande construido. Sólo unos cuantos cambios y adaptaciones han sido necesarios para hacer el sistema UNIX utilizable sobre los nuevos micro-ordenadores, y hay un acuerdo general en que no existe otro sistema operativo más portable. El valor de esta portabilidad no puede ser sobreestimado, porque el desarrollo software es caro y tedioso. Generalmente se está de acuerdo en que el sistema UNIX proporciona el entorno requerido para permitir el fácil traslado de las aplicaciones desde los micro-ordenadores a los maxi-ordenadores.

■ *Flexibilidad.* Un atractivo importante del sistema UNIX para los creadores de software y hardware es su flexibilidad. El UNIX ha sido adaptado a aplicaciones tan divergentes como la automatización de fábricas, los sistemas de conmutación telefónica y los juegos personales. Se han ido añadiendo nuevas funciones y órdenes a paso rápido, y la mayoría de los creadores manifiestan su preferencia por el sistema UNIX como “banco de trabajo” para sus aplicaciones.

■ **Potencia.** UNIX es uno de los sistemas operativos más potentes disponibles para cualquier computador. Su sintaxis de órdenes clara y concisa permite a los usuarios hacer muchas cosas rápida y sencillamente. Los usuarios pueden aprovechar los servicios y órdenes internos del sistema UNIX que serían añadidos caros en otros sistemas. Ningún sistema operativo es más rico en capacidades que el UNIX.

■ **Multiusuario y multitarea.** Debido a que el sistema UNIX es un entorno multitarea de tiempo compartido, puede hacer más de una cosa a la vez fácilmente. En un sistema UNIX personal, un usuario puede estar editando un fichero, imprimiendo otro fichero sobre una impresora, enviando correo electrónico a otra máquina y utilizando una hoja de cálculo electrónica, simultáneamente. El sistema UNIX está diseñado para manejar sin esfuerzo las necesidades múltiples y simultáneas de un usuario. También es un entorno multiusuario, que soporta las actividades de más de una persona a la vez. No es infrecuente en versiones del sistema UNIX sobre grandes unidades centrales soportar varios cientos de usuarios a la vez, y todos estos usuarios tienen la misma “visión privada” del sistema que tiene un solo usuario sobre un microordenador.

■ **Elegancia.** El sistema UNIX está ampliamente considerado como uno de los sistemas operativos más elegante. Una vez que los usuarios comprenden algunos de los conceptos básicos del sistema UNIX, pueden realizar muchas y grandes tareas de un modo sencillo. Los creadores de otros sistemas operativos y otras aplicaciones con frecuencia toman prestadas ideas y temas del sistema UNIX para enriquecer sus propios sistemas.

Todas estas razones ayudan a contabilizar el rápido empuje de popularidad que el sistema UNIX ha gozado en estos años. Estos factores dejan claro que el sistema continuará creciendo y desarrollándose, y la mayoría de los usuarios de ordenadores se lo encontrarán finalmente en un sitio u otro. Muchos llegarán a disponer de su propio sistema UNIX sobre su microordenador personal.

1.1.- Aproximación a UNIX

El sistema UNIX sigue siendo en cierto modo controvertido, aunque sus capacidades multiusuario y multitarea están llegando a ser cada vez más ampliamente respetadas en la comunidad de microordenadores.

La filosofía original “hacer una cosa bien” del sistema UNIX primitivo se ha perdido en cierta manera en las versiones modernas, y las órdenes actuales contienen un gran número de opciones y controles. Esto es una bendición a medias, porque ahora las órdenes pueden ser más difíciles de usar. Sin embargo, también ellas pueden ser adaptadas a nuevas situaciones sin contorsiones. No obstante, los conceptos clave de la filosofía primigenia, tal como los cauces, han sido retenidos, mientras las capacidades de las órdenes han sido ampliadas para satisfacer nuevas necesidades.

El sistema UNIX está orientado principalmente a terminales *basados-en-carácter*. Se necesita un software especial para hacer que el sistema UNIX funcione con pantallas gráficas de mapa de bits.

UNIX va contra la tendencia actual de hacer los sistemas operativos “invisibles” al usuario. Muchos sistemas operativos han intentado crear interfaces de usuario que eliminan el sistema operativo de la vista del usuario. La interfaz de usuario orientada a ventanas y menús ayuda a ocultar las órdenes, sistemas de ficheros y herramientas administrativas del usuario. Al utilizar el sistema UNIX, sin embargo, mientras más consciente se sea de las funciones internas del sistema, mejor se podrá controlar para nuestro propio beneficio y para mejorar su productividad. Este hecho se deriva principalmente de su creación como sistema destinado a expertos, en el que la mayoría de los usuarios estaban bien al tanto de las interioridades del sistema que utilizaban. Ahora que el sistema UNIX ha sido adoptado por una comunidad de usuarios más amplia, se han dispuesto herramientas tales como agentes de usuario para aislar a éste de muchas de las complejidades del sistema operativo. Sin embargo, aun cuando se puede utilizar ahora el sistema sin una comprensión extensa de sus complejidades, podemos beneficiarnos sustancialmente de un mejor conocimiento acerca de lo que estamos haciendo realmente cuando utilizamos una orden.

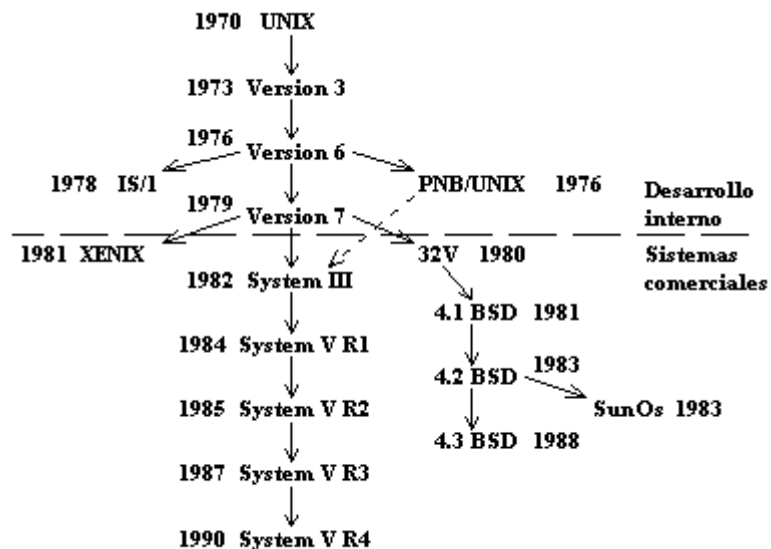
1.2.- Historia del sistema UNIX

A pesar de los sistemas abiertos, la historia de UNIX está dominada por el ascenso y caída de los sistemas hardware. UNIX nació en 1969 en una mainframe 635 de Genaral Electric. A la vez, los Laboratorios Bell de AT&T había completado el desarrollo de Multics, un sistema multiusuario que falló por su gran demanda de disco y memoria. En respuesta a Multics, los ingenieros de sistemas Kenneth Thompson y Dennis Ritchie inventaron el UNIX. Inicialmente, Thompson y Ritchie diseñaron un sistema de archivos para su uso exclusivo, pero pronto lo cargaron en una Digital Equipment Corp. (DEC) PDP-7, una computadora con solo 18 kilobytes de memoria. Este suministraba una larga serie de puertos. En 1970, fue cargado en una PDP-11, y el runoff, el predecesor del troff, se convirtió en el primer procesador de texto de UNIX. En 1971, UNIX recibió reconocimiento oficial de AT&T cuando la firma lo usó para escribir manuales.

La segunda edición de UNIX fue realizada en 1971. La segunda edición dio forma al UNIX moderno con la introducción del lenguaje de programación C y sobre los 18 meses siguientes, el concepto de los pipes. Los pipes fueron importantes por muchas razones. Representaron una nueva forma de tratamiento de datos. Desde un punto de vista moderno, los pipes son un mecanismo orientado a objetos, porque entregan datos desde un objeto, o programa, a otro objeto.

Mientras tanto, había mucha actividad con el C. El C es otro producto de los Laboratorios Bell. Fue formado a partir de conceptos de otros tres lenguajes: B, CPL (Combined Programming Language) y Algol-60. A finales de 1973, después de que Ritchie añadió soporte para variables globales y estructuras, C se convirtió en el lenguaje de programación de UNIX preferente. (Brian Kernigham, quien

ayudó a Ritchie a desarrollar el C, añadió la R al estándar K&R, es el estándar preferido hasta la aceptación del ANSI C).



La tabla 1 muestra una concisa historia del desarrollo de UNIX.

El ascenso del C fue responsable del concepto de portabilidad. Escrito en C, el entorno UNIX pudo ser relativamente fácil de trasladar a diferentes plataformas hardware. Las aplicaciones escritas en C pudieron ser fáciles de transportar entre diferentes variantes de UNIX. En esta situación nació el primer criterio de sistema abierto: portabilidad OS, la posibilidad de mover software desde una plataforma hardware a otra de una forma estándar. La portabilidad de UNIX se convirtió en el modelo de transportar aplicaciones en C desde un sistema UNIX a otro.

En 1974, la quinta edición de UNIX fue realizada para que estuviera disponible para las universidades. El precio de la versión 5 fue suficiente para recuperar los costos de las cintas y manuales. Se informó de los errores directamente a Thompson y Ritchie, quienes los reparaban a los pocos días de la notificación. En 1975, la sexta edición de UNIX fue desarrollada e iniciada para ser ampliamente usada. Durante este tiempo, los usuarios se hicieron activos, los grupos de usuarios fueron formados, y en 1976, se estableció el grupo de usuarios USENIX. En 1977, Interactive System Corp. inició la venta de UNIX en el mercado comercial. Durante este tiempo, UNIX también adquirió más poder, incluyendo soporte para procesadores punto flotante, microcódigo y administración de memoria.

Con la creciente popularidad de los microprocesadores, otras compañías trasladaron el UNIX a nuevas máquinas, pero su simplicidad y claridad tentó a muchos a aumentarlo bajo sus puntos de vista, resultando muchas variantes del sistema básico. En el período entre 1977 y 1982, los Laboratorios Bell combinaron algunas variantes de AT&T dentro de un sistema simple, conocido comercialmente como UNIX System III. Los Laboratorios Bell más tarde añadieron muchas

características nuevas al UNIX System III, llamando al nuevo producto UNIX System V, y AT&T anunció su apoyo oficial al System V en Enero de 1983. Sin embargo, algunas personas en la Universidad de California en Berkeley habían desarrollado una variante del UNIX, BSD, para máquinas VAX, incluyendo algunas nuevas e interesantes características.

A comienzos de 1984, había sobre 100.000 instalaciones del sistema UNIX en el mundo, funcionando en máquinas con un amplio rango de computadoras, desde microprocesadores hasta mainframes. Ningún otro sistema operativo puede hacer esta declaración. Muchas han sido las razones que han hecho posible la popularidad y el éxito del sistema UNIX:

- El sistema está escrito en un lenguaje de alto nivel, haciéndolo fácil de leer, comprender, cambiar, y mover a otras máquinas. Ritchie estimó que el primer sistema en C era de un 20 a un 40 por ciento más grande y lento porque no estaba escrito en lenguaje ensamblador, pero las ventajas de usar un lenguaje de alto nivel superaban largamente a las desventajas.
- Posee una simple interface de usuario con el poder de dar los servicios que los usuarios quieren.
- Provee de primitivas que permiten construir programas complejos a través de programas simples.
- Usa un sistema de archivos jerárquico que permite un mantenimiento fácil y una implementación eficiente.
- Usa un formato consistente para los archivos, el flujo de bytes, haciendo a los programas de aplicación más fáciles de escribir.
- Provee una simple y consistente interface a los dispositivos periféricos.
- Es un sistema multiusuario y multitarea; cada usuario puede ejecutar varios procesos simultáneamente.
- Oculta la arquitectura de la máquina al usuario, haciendo fácil de escribir programas que se ejecutan en diferentes implementaciones hardware.

Sin embargo tiene algunos inconvenientes:

- Comandos poco claros y con demasiadas opciones.
- Escasa protección entre usuarios.
- Sistema de archivo lento.

A pesar de que el sistema operativo y muchos de los comandos están escritos en C, UNIX soporta otros lenguajes, incluyendo Fortran, Basic, Pascal, Ada, Cobol, Lisp y Prolog. El sistema UNIX puede soportar cualquier lenguaje que tenga un compilador o intérprete y una interface de sistema que defina las peticiones del usuario de los servicios del sistema operativo de la forma estándar de las peticiones usadas en los sistemas UNIX.

Año	Evento	Descripción
1965	Origen	Bell Telephone Laboratories y General Electric Company intervienen en el proyecto MAC (del MIT) para desarrollar MULTICS.
1969-71	Infancia del UNIX	El primer UNIX llamado Versión 1 o Primera edición, nace de las cenizas de MULTICS.
1972-73	Nace el C	En la Versión 2 el soporte del lenguaje C y los pipes son añadidos. En la Versión 4 el ciclo se completa con la reescritura de UNIX en C.
1974-75	El momento	Las Versiones 5 y 6 de UNIX se distribuyen a las universidades. La Versión 6 circula en algunos ambientes comerciales y gubernamentales. AT&T impone ahora pagar una licencia, a pesar de que no puede promocionar UNIX por las duras regulaciones de EEUU del monopolio telefónico de AT&T.
1977	UNIX como producto	Interactive Systems es la primera compañía comercial que ofrece UNIX.
1977	Nace BSD	1BSD incluye un Shell Pascal, dispositivos y el editor ex.
1979	Versión 7	La Versión 7 de UNIX incluye el compilador completo K&R con uniones y definiciones de tipos. Versión 7 también añade el Bourne Shell.
1979	Trabajo en Red	BSD acrecentado por BBN incluye soporte para trabajar en red.
1979	Nace XENIX	Implementación para microcomputadoras ampliamente distribuido en hardware de bajo coste.

1980	Memoria Virtual	La capacidad de memoria virtual se añade en 4BSD.
1980	Nace ULTRIX	DEC realiza una versión de UNIX basado en BCD.
1980	Licencias en AT&T	La distribución de licencias abre el mercado.
1982	Atracción de los negocios	Soporte importante para procesos de transacciones desde UNIX System Development Lab.
1983	Nace System V	La versión más común de AT&T obtiene sus bases.
1984	Salida de SVR3	AT&T desata la versión más popular de System V hasta ahora.
1988	Motif vs Open Look	Sistemas por ventanas rivales son anunciados por OSF y UI.
1988	Siguiente paso	Un UNIX gráfico usa el Kernel Mach.
1990	OSF/1 vs SVR4	Versiones rivales de UNIX son anunciadas por OSF y UI.
1992-95	Socialización	OSF/1 abandona la escena; SVR4 se convierte en el estándar; Sun vende más estaciones de trabajo para usuarios de Motif que para usuarios de Open Windows; y crece Windows/NT de Microsoft.

Tabla 1. Hechos importantes en la historia de UNIX.

1.3.- Los Diferentes Sistemas UNIX

La estandarización de UNIX se ha convertido en un tema cada vez más debatido. Parece poco probable que en el futuro surja una norma UNIX única. AT&T continua promoviendo su versión llamada UNIX System V, muy utilizada en la industria. Por otro lado, las universidades siguen promoviendo la versión UNIX de Berkeley, el cual es un derivado de la versión de AT&T.

La comunidad UNIX ha cooperado en el desarrollo de una especificación estandarizada del sistema denominada POSIX, que consiste de un subconjunto común de los principales sistemas UNIX. La fundación de software abierto se constituyó para producir una versión de UNIX basada, en gran medida, en la versión AIX de IBM.

Pasarán muchos años antes de que aparezca solo UNIX estandarizado, si es que se consigue alguna vez. Tal vez no exista un diseño de sistemas operativos capaz de satisfacer las diversas necesidades de la comunidad informática mundial.

El origen de los diferentes sistemas UNIX tiene su raíz en lo que es el nacimiento, en 1975 de la versión 6 de los laboratorios AT&T de los Laboratorios Bell. Después de la presentación de esta versión dos líneas diferentes conocidas como Sistema V y BSD.

Los desarrolladores de la Universidad de California en Berkeley (de ahí el nombre de BSD) han agrandado UNIX de diferentes formas añadiendo un mecanismo de memoria virtual, el shell C, el control de tareas, la red TCP/IP, por nombrar solo un pequeño número. Algunas de estos nuevos mecanismos fueron introducidos en las líneas de código de AT&T.

El sistema V versión 4 es presentado como la fusión del Sistema V y de BSD, pero eso no es completamente exacto. El sistema V Versión 4 resulta de la incorporación de las funciones más importantes de BSD y de SunOS en el seno de Sistema V. Esta unión puede ser vista como una unión más que como una fusión, en la cual algunas características de cada uno son heredadas (a las cuales se debe añadir características cuyo origen es incierto).

La proliferación de constructores informáticos en el curso de los años 80's provocó la aparición en el mercado de decenas de nuevos sistemas UNIX. UNIX fue escogido por su bajo costo y por sus características técnicas, pero también a causa de la ausencia de otras opciones. Estos proveedores se basaron en versiones de BSD aportando modificaciones menores y/o más importantes. La mayor parte de los que aún subsisten provienen del sistema V versión 3 (en general versión 3.2), sistema V versión 4 y algunas veces de BSD 4.2 o 4.3 (SunOS es una excepción ya que tiene su origen en una versión más antigua de BSD). Para complicar las cosas, varios proveedores han mezclado características de BSD y del Sistema V en el corazón de un solo sistema operativo.

2.1 El sistema XENIX

XENIX es la primera versión de UNIX diseñada para microcomputadoras, aún es utilizada. Esta versión proviene de la versión 7 y ha sido convertido progresivamente en un sistema V versión 2.

XENIX influenció Sistema V versión 3, la mayor parte de estas funciones fueron incorporadas en el Sistema V versión 3.2

2.2 El sistema OSF/1

En 1988, Sun y AT&T se pusieron de acuerdo para desarrollar juntos las futuras versiones de Sistema V. En respuesta, IBM, DEC, Hewlett-Packard así como otros constructores y sociedades informáticas fundaron la OSF (Open Software Foundation) cuyo objetivo era la concepción de otro sistema operativo compatible con UNIX y, sobre todo, independiente de AT&T.

OSF/1 es el resultado de este esfuerzo, aunque OSF/1 constituye más una definición de estándares que una implementación real.

Entre los estándares más importantes se encuentran POSIX (definido por IEEE/ANSI), el AT&T System V Interface Definition (SVID), la Application Environment Specification (AES) de la OSF y el X/Open Portability Guide de la X/Open, un consorcio fundado en Gran Bretaña en 1984.

2.3 El sistema SCO UNIX

Este nombre hace referencia a SCO Open Desktop y SCO Open Server Release 3 producidos por Santa Cruz Operations Inc. (que funciona sobre procesadores 486). Este sistema operativo es una implementación de V.3.2.5.

2.4 El sistema SunOS

Es el sistema operativo de tipo BSD más conocido que ha introducido, en el mundo UNIX, funcionalidades importantes (entre la más importante esta NFS). Sun a querido reemplazar SunOS por Solaris pero ha cedido a la presión de los usuarios: Sun continúa proporcionando los dos sistemas operativos.

2.5 El sistema Solaris

Es una implementación del sistema V.4 propuesto por Sun. Hay que mencionar que Solaris 2.x a veces es denominado SunOS 5.x.

2.6 El sistema HP-UX

Es la versión de UNIX de Hewlett-Packard que sigue las características del Sistema V incorporando varias características de OSF/1. HP-UX ha sido considerablemente modificado entre las versiones 9 y 10. Desde el punto de vista de la administración, HP-UX 9 se parece al sistema V.3 con algunas extensiones, por otro lado HP-UX 10 se asemeja a un sistema operativo del tipo V.4.

2.7 El sistema DEC OSF/1

La versión OSF/1 de Digital Equipment Corporation se parece en gran medida a un sistema BSD genérica del punto de vista de la administración del sistema, aunque en el fondo se trata de un Sistema V. HP-UX y DEC OSF/1 claman su conformidad a un conjunto de estándares prácticamente idénticos pero estas versiones deben ser administradas de forma diferente.

2.8 El sistema IRIX

Las primeras versiones de IRIX incorporan numerosas características de BSD pero estas han desaparecido en el transcurso del tiempo a favor de una conformidad a V.4.

2.9 El sistema AIX

El sistema operativo de IBM de tipo Sistema V, también ofrece diferentes funcionalidades de V.4, BSD y OSF/1 (además de las inevitables características propias a IBM).

2.10 El sistema UNIX

UNIX es un clon de UNIX en el dominio público destinado a los procesadores Intel. UNIX ha ganado en popularidad regularmente y es muy útil en varias situaciones: es un sistema UNIX poco costoso que puede constituir un ambiente de investigación para los colegios y universidades, una solución económica para contar con una conexión Internet para las empresas pequeñas, un sistema UNIX doméstico para los profesionales y una terminal X barata para los sitios UNIX con presupuesto reducido.

El núcleo fue desarrollado por Linus Torvalds, (UNIX es el UNIX de Linus, Linus UNIX) aunque otras personas han contribuido (y contribuyen) a su desarrollo. UNIX es globalmente de tipo BSD.

Técnicamente, el nombre de UNIX hace referencia al corazón del sistema operativo (el núcleo y algunos controladores de periféricos) pero ese nombre también se aplica al software de dominio público, donde las fuentes son de origen variado, que constituyen una distribución.

2.11 El sistema MINIX

MINIX es un sistema operativo desarrollado por Andrew Tanenbaum con fines pedagógicos. Pensado en un principio para ser ejecutado a partir de discos flexibles, en una PC compatible. MINIX fue la fuente de inspiración de Linus para desarrollar el sistema operativo UNIX.

2.12 El sistema FreeBSD

FreeBSD es un sistema operativo UNIX BSD avanzado para arquitecturas Intel (x86), DEC Alpha y PC-98. Es atendido por un gran equipo de personas repartidas en todo el mundo.

2.13 El sistema OpenBSD

El proyecto OpenBSD produce una multiplataforma libre del sistema operativo UNIX 4.4 BSD. Los esfuerzos de los integrantes del proyecto van dirigidos a reforzar la portabilidad, estandarización, seguridad, *correctness* e

programa deseado. También es un lenguaje de programación de alto nivel que puede utilizarse en la combinación de programas de utilidad para crear aplicaciones completas.

El shell puede soportar múltiples usuarios, múltiples tareas, y múltiples interfaces para sí mismo. Los dos shells más populares son el BourneShell (System V) y el Cshell (BSD UNIX), debido a que usuarios diferentes pueden usar diferentes shells al mismo tiempo, entonces el sistema puede aparecer diferente para usuarios diferentes. Existe otro shell conocido como KornShell (así llamado en honor de su diseñador), que es muy popular entre los programadores.

Utilerías: El Sistema Operativo UNIX incluye una gran variedad de programas de utilidad que pueden ser fácilmente adaptadas para realizar tareas específicas. Estas utilerías son flexibles, adaptables, portables y modulares, y pueden ser usadas junto con filtros y redireccionamientos para hacerlos más poderosos.

Sistema Multiusuarios: Dependiendo del equipo disponible, un UNIX puede soportar desde uno hasta más de 100 usuarios, ejecutando cada uno de ellos un conjunto diferente de programas.

Sistema Multitareas: UNIX permite la realización de más de una tarea a la vez. Pueden ejecutarse varias tareas en su interior, mientras se presta toda la atención al programa desplegado en la terminal.

Estructura de Archivos: La estructura de archivos del UNIX está pensada para facilitar el registro de una gran cantidad de archivos. Utiliza una estructura jerárquica o de árbol que permite a cada usuario poseer un directorio principal con tantos subdirectorios como desee; UNIX también permite a los usuarios compartir archivos por medio de enlaces (links), que hacen aparecer los archivos en más de un directorio de usuario.

Además, UNIX permite proteger los archivos del usuario contra el acceso por parte de otros usuarios.

Entrada y Salida Independiente del Dispositivo: Los dispositivos (como una impresora o una terminal) y los archivos en disco son considerados como archivos por UNIX. Cuando se da una instrucción al UNIX puede indicársele que envíe el resultado a cualquiera de los diversos dispositivos o archivos. Esta desviación recibe el nombre de redireccionamiento de la salida.

En forma similar, la entrada de un programa puede redireccionarse para que venga de un archivo en disco. En el UNIX, la entrada y la salida son independientes del dispositivo, pueden redireccionarse hacia o desde cualquier dispositivo apropiado.

Comunicación Entre Procesos: UNIX permite el uso de conductos y filtros en la línea de comandos. Un conducto (pipe) redirige la salida de un programa para que se convierta en entrada de otro. Un filtro es un programa elaborado para procesar un flujo de datos de entrada y producir otro de datos de salida. Los conductos y filtros suelen usarse para unir utilerías y realizar alguna tarea específica.

1.5.- Estructura del Sistema

La figura 1 describe la arquitectura de alto nivel de UNIX. El sistema operativo interactúa directamente con el hardware, suministrando servicios comunes a los programas y aislándolos de la particularización del hardware. Viendo el sistema como un conjunto de capas, el sistema operativo es comúnmente llamado como núcleo del sistema o kernel. Como los programas son independientes del hardware que hay por debajo, es fácil moverlos desde sistemas UNIX que corren en diferentes máquinas si los programas no hacen referencia al hardware subyacente. Por ejemplo, programas que asumen el tamaño de una palabra de memoria será más difícil de mover a otras máquinas que los programas que no lo asumen.

Los programas como el shell y los editores (ed y vi) mostrados en la capa siguiente interactúa con el kernel invocando un conjunto bien definido de llamadas al sistema. Las llamadas al sistema ordenan al kernel realizar varias operaciones para el programa que llama e intercambiar datos entre el kernel y el programa. Varios programas mostrados en la figura 1 están en configuraciones del sistema estándares y son conocidos como comandos, pero los programas de usuario deben estar también en esta capa, indicándose con el nombre a.out, el nombre estándar para los archivos ejecutables producidos por el compilador de C. Otros programas de aplicaciones pueden construirse por encima del nivel bajo de programas, por eso la existencia de la capa más exterior en la figura 1. Por ejemplo, el compilador de C estándar, cc, está en el nivel más exterior de la figura: invoca al preprocesador de C, compilador, ensamblador y cargador, siendo todos ellos programas del nivel inferior. Aunque la figura muestra una jerarquía a dos niveles de programas de aplicación, los usuarios pueden extender la jerarquía a tantos niveles como sea apropiado. En realidad, el estilo de programación favorecida por UNIX estimula la combinación de programas existentes para realizar una tarea.

Muchos programas y subsistemas de aplicación que proporcionan una visión de alto nivel del sistema tales como el shell, editores, SCCS (Source Code Control System) y los paquetes de documentación, están convirtiéndose gradualmente en sinónimos con el nombre de "Sistema UNIX". Sin embargo, todos ellos usan servicios de menor nivel suministrados finalmente por el kernel, y se aprovechan de estos servicios a través del conjunto de llamadas al sistema. Hay alrededor de 64 llamadas al sistema en System V, de las cuales unas 32 son usadas frecuentemente. Tienen opciones simples que las hacen fáciles de usar pero proveen al usuario de gran poder. El conjunto de llamadas al sistema y los algoritmos internos en los que se implementan forman el cuerpo del kernel. En resumen, el kernel suministra y define los servicios con los que cuentan todas las aplicaciones del UNIX.

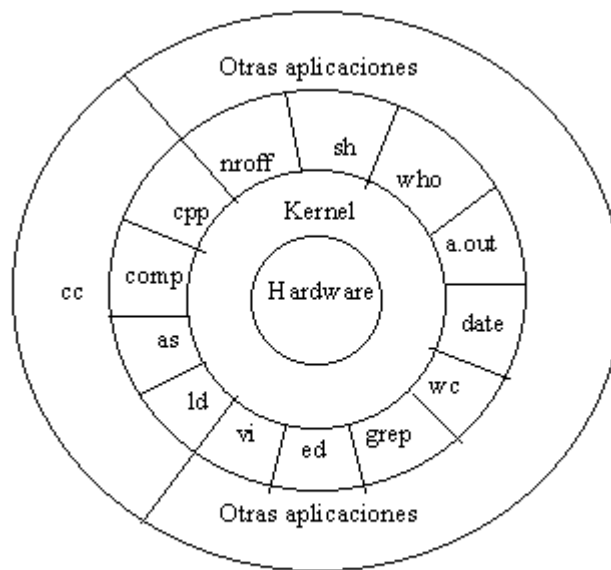
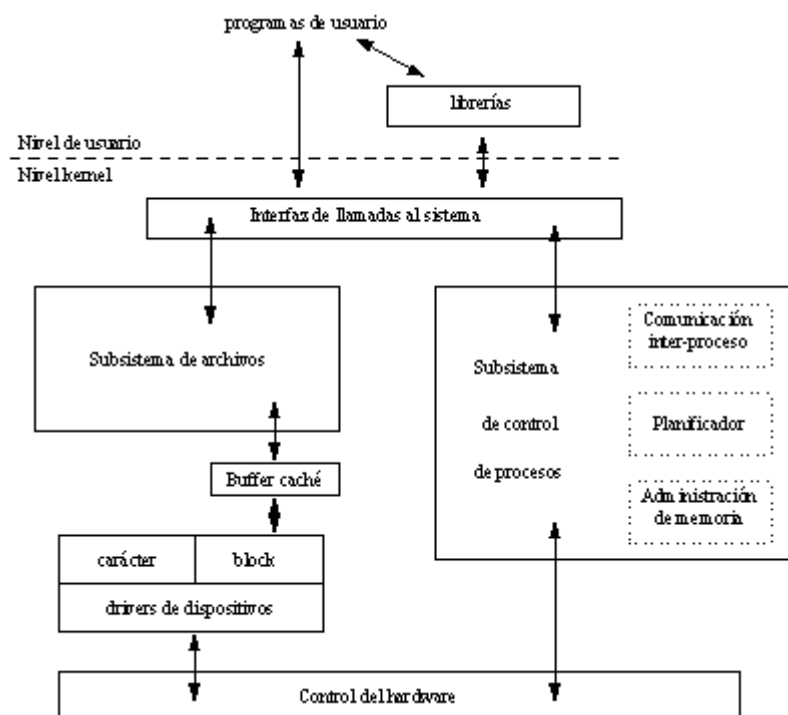


Figura 2.1 Arquitectura del sistema UNIX.



1.6.- Objetivos de UNIX

UNIX fue diseñado teniendo en mente los siguientes objetivos:

- crear un sistema interactivo de tiempo compartido diseñado por programadores y para programadores, destinado a usuarios calificados.

- que fuera sencillo, elegante, escueto y consistente.
- que permitiera resolver problemas complejos combinando un número reducido de comandos básicos.

1.7.- Filosofía de UNIX

Los objetivos con que se creó determinaron una "filosofía" de UNIX caracterizada por:

- comandos cortos, simples, específicos y muy eficientes, que "hacen una sola cosa pero la hacen muy bien".
- entrada y salida estandarizadas que permiten la interconexión de comandos. Esto se llama entubamiento ("pipelining"): la salida de un comando es tomada por el siguiente como entrada.

1.8.- El UNIX moderno

Orientado en primera instancia a terminales de caracteres, actualmente dispone de la interface gráfica X-Windows. Esto ha simplificado mucho el uso para los no especialistas.

Es ideal para trabajar como servidor: una máquina potente como servidor, terminales o computadores personales baratas en los puestos de trabajo. El paquete de libre uso Samba permite que una máquina UNIX actúe como servidor de puestos de trabajo Windows 3.11, Windows 95, y Windows NT.

Se orienta en la dirección contraria a la tendencia de hacer invisible al usuario el sistema operativo, permitiendo el uso de todas las bibliotecas, llamadas al sistema y herramientas internas, aunque su uso requiere un alto nivel de especialización. Es posible compilar un núcleo específicamente adaptado a las necesidades particulares de una empresa o grupo de trabajo, reduciendo el uso de recursos y aumentando la rapidez.

Las máquinas RISC de los '80 y '90 adoptaron UNIX como sistema operativo; es por lejos el sistema predominante en las estaciones de trabajo orientadas a cálculos e investigación. También fue adoptado para la creación de Internet, mayoritariamente soportada por UNIX.

Por sus características de diseño, está especialmente preparado para su ampliación y desarrollo en 64 bits y el multiprocesamiento en varias CPUs.

Inicio de sesión en UNIX

Antes de adentrarnos en explicar cómo podemos acceder a un sistema UNIX, vamos a hacer algunas consideraciones necesarias como pequeña introducción:

☞ Un mismo usuario puede conectarse simultáneamente en varios terminales, así como tener asignados varios nombres con características de entorno diferentes.

☞ El *Administrador del sistema* asigna los nombres de usuario y suele proporcionar una primera clave de acceso. El usuario puede y debe cambiarla periódicamente. El administrador sólo puede eliminarla en un momento dado, pero no descubrir su contenido. Los caracteres introducidos en la clave no se visualizan en pantalla. Conviene ser cuidadoso al teclearla, evitando correcciones. En caso de clave errónea el sistema muestra un mensaje de advertencia y vuelve a solicitar la identificación. En los casos de cambio de clave, se solicita confirmación. Se obliga a repetir la introducción de la clave para evitar posibles errores inadvertidos. Hay que tener en cuenta que el UNIX es sensible a la diferencia entre minúsculas y mayúsculas.

☞ En el inicio de sesión (al solicitar la identificación de usuario o proceso “*login in*”), el sistema aún no conoce las características del terminal. Por ello conviene usar minúsculas. Si se empieza utilizando mayúsculas, el sistema interpreta que el terminal sólo admite mayúsculas, por lo que aceptará todas las órdenes en mayúsculas, aunque internamente las ejecutará todas en minúsculas.

☞ Al inicio, el sistema tampoco reconocerá teclas habituales de borrado, corrección o anulación; en estos casos es aconsejable pulsar dos veces la combinación [**Ctrl.** + **d**] y reiniciar el proceso de acceso.

☞ El “*prompt*” o indicador de órdenes es la forma que tiene el sistema, y, más concretamente su intérprete de órdenes o “*shell*”, de indicar al usuario su disposición para recibir órdenes. Hasta que no aparezca este símbolo no podremos dar órdenes al sistema. El símbolo habitual del prompt es **\$**, correspondiente al shell Bourne (sh) y al shell Korn (ksh). El shell C (csh) utiliza el carácter **%**. El símbolo **#** se reserva para el acceso como administrador (superusuario o *root*). De esta forma puede reconocerse a

través de los símbolos el tipo de shell en el que se está trabajando, y si se ha accedido como usuario normal o como administrador del sistema.

☞ UNIX proporciona varios cientos de órdenes o utilidades, aunque no todas estarán disponibles en todos los sistemas. Su número aumenta constantemente y cada usuario suele construirse algunas propias.

☞ En cualquier momento se dispone de dos utilidades de ayuda: **man** y **help**.

☞ Conviene resaltar la necesidad de abandonar correctamente la sesión con la orden **exit**. Esto evita el acceso posterior por usuarios ajenos a nuestros archivos de trabajo, la facturación por tiempos excesivos, y contribuye al buen mantenimiento y organización del sistema.

☞ Hay que recordar que no basta con apagar el terminal. Es preciso comunicar al sistema la salida y, posteriormente, desconectar.

2.1.- Acceso a un Sistema UNIX

UNIX es un SO multiusuario y multitarea. Puede arrancar en seis niveles diferentes (ver más adelante el apartado dedicado a la secuencia de arranque, para más información). Así, puede arrancar en modo monousuario, por ejemplo, para labores de administración del sistema. Sin embargo, lo más común es el arranque en modo multiusuario. Incluso, se puede acceder a él desde “*terminales tontos*” o a través de otros SO, mediante TELNET. De cualquier forma, al iniciar cualquier sesión en UNIX debemos tener presente que sólo podremos acceder a los servicios del sistema si tenemos un usuario creado dentro de él. Dentro de UNIX encontramos 2 tipos de usuarios, de los que nos ocuparemos más adelante. Se trata de los usuarios normales y del *Superusuario* o *root*.

Cada uno de los usuarios se identifica externamente por un *login* y un *password*, e, internamente, por un identificativo de sistema (UID). Asimismo, cada usuario pertenece a un grupo determinado, que, internamente, también se identifica por un número de grupo (GID). Para más información acerca de usuarios y grupos, remitimos a la sección dedicada a ellos dentro de Administración de Usuarios.

El terminal usado para conectar al sistema UNIX podrá ser una consola del sistema, un terminal orientado a carácter conectado directamente a un puerto RS232 en la máquina, un terminal remoto conectado vía línea telefónica o módem, o un emulador de terminal usado el servicio TELNET. No importa cuál de ellos estemos utilizando, ya que el sistema actuará de igual forma para todos estos tipos de terminales.

Ahora, sentados frente a la terminal con acceso a UNIX, nos aparece una pantalla muy graciosa donde nos pide que nos identifiquemos ante el sistema. De repente vemos ante nosotros algo similar a:

Bienvenido al sistema EL_J_Y_EL_J

login:

No!!!!!!!, no te asustes, no te has equivocado y has entrado, por error en el sistema de la Nasa. UNIX nos pide que nos identifiquemos para saber si tenemos cuenta dentro del sistema. Es una medida de seguridad para no permitir el acceso a cualquier persona a los recursos y servicios que ofrece cada servidor. Ahora introduce tu nombre de usuario. Recuerda que UNIX es un sistema *case sensitive*, es decir, y para que se entienda, que hace distinción entre mayúsculas y minúsculas. Así, “Remigio” será distinto de “remigio” y también de “REMIGIO”. Todos los ID deberían tener un carácter minúscula al principio (si tu ID comienza por mayúscula, el sistema pensará que estás accediendo desde un terminal que sólo tiene mayúsculas y nos tratará de forma diferente). El ID de presentación será único, es decir, ningún otro usuario tendrá el mismo ID que tú

Ahora ya has introducido tu *login* “remigio. Lleva cuidado con no meter la pata cuando teclees tu login, ya que no podrás corregir los errores con la tecla BACKSPACE. Esto, aunque a primera vista parezca una solemne estupidez, se debe a que el sistema aún no tiene ni idea de quién eres, aún no has sido identificado. El uso de la tecla BACKSPACE para borrar un carácter se considera una preferencia del usuario.

Después de haberte identificado (suponemos que correctamente, que nos conocemos...), el sistema solicitará ahora tu “palabra de paso” individual (*password* la llaman los americanos).

Bienvenido a El_J_y_El_J

login: remigio

password:

El sistema quiere que demuestres ser la persona autorizada para utilizar ese ID de presentación (sí, ya sé que me vas a decir que no habrá muchos “remigios”; pero suponte que en su lugar te hubieras llamado “pepe”). Después de todo, lo único que quiere hacer el sistema es proteger tus ficheros y otros datos privados frente a los demás (por ejemplo, tus números de la primitiva que tan celosamente guardas en un archivo...). Por tanto, introduce tu contraseña individual, finalizando con RETURN.

Tranquilo!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!, no es un error del sistema. Te creo. Sé que estás escribiendo tu password, aunque no aparezca nada en la pantalla. El sistema está haciendo *ew*. Es una medida de seguridad que impide que alguien que esté mirando por encima de tu hombro pueda descubrir la contraseña (como tu jefe, que quiere quitarte los números de la primitiva).

Si tecleas incorrectamente tu ID de presentación o tu contraseña, o si no eres un usuario autorizado (pillín, intentando entrar donde no debes...), el sistema responderá del modo siguiente:

Bienvenido a El_J_y_El_J

Login: remigio

Password:

Login incorrect

Login:

El sistema, que quiere hacerte pasarlo mal, no te dirá si el ID de presentación fue introducido incorrectamente o si la contraseña estaba equivocada. Lo que hará ahora es proporcionarte una segunda oportunidad (intenta no fastidiarla ahora...). Cuando el ID y la contraseña hayan sido aceptados, aparecerán en pantalla algunos mensajes del sistema. Ya hemos conseguido entrar. Remigio, has dado el primer paso para convertirte en un verdadero hacker...

2.2.- Consideraciones sobre acceso a un sistema UNIX

A continuación te proporcionaremos algunas consideraciones que debes tener en cuenta antes de acceder a cualquier sistema UNIX. Veamos:

✎ Sólo tendrán acceso al sistema aquellos usuarios a los que el administrador (sí, ese tío que se pasea creyendo saberlo todo), haya asignado previamente un identificador UID.

✎ El identificador de usuario asignado por el administrador puede tener longitud variable, pero sólo serán significativos los primeros 8 caracteres (así que olvídate de poner como identificador el himno del Escalerillas C. F.). No están permitidos los espacios en blanco.

✎ Para incrementar la seguridad, cada usuario puede añadir, opcionalmente, una contraseña de acceso al sistema. La existencia de una contraseña puede ser impuesta por el administrador.

✎ Cuando se va a trabajar desde un terminal conectado directamente al ordenador, nada más encenderlo aparecerá un mensaje solicitando la identificación de usuario. Este mensaje puede ser modificado por el administrador.

✎ Siempre que el sistema esté en espera para recibir órdenes, aparecerá en la pantalla el *prompt* (¿eso qué es? Sí, sé que de eso no hemos hablado aún, pero no desesperes, en nada te lo explico).

✎ Para finalizar una sesión de trabajo basta con pulsar **[ctl.+d]** o teclear 'exit'. Una vez finalizada la sesión, el sistema vuelve a mostrar los mensajes con los que solicita el login y la password. Más adelante aprenderás a apagar el sistema tú solito (a ver, ¿acaso te has creído que esto es como tu Windows 95?).

Una vez que haya aparecido el shell del sistema ya podremos teclear órdenes. El shell que nos proporciona es el Shell Korn, que nos permite editar las órdenes escritas con anterioridad, aunque no de una forma tan fácil como en DOSKEY del MS-DOS. Para pasar a modo de edición hay que pulsar <ESC>, y podemos usar las siguientes teclas:

K : muestra las órdenes anteriores.

J : muestra las órdenes posteriores.

I : modo inserción delante del carácter en que se encuentra el cursor.

A : inserción a continuación del carácter en que está el cursor.

X : borra el carácter en el que está el cursor.

R : sustituir un carácter.

ESPACIO : desplazarse hacia delante en la línea que estamos editando.

← : desplazarse hacia atrás en la línea que estamos editando.

Arranque y Desconexión

Generalmente hay varios procesos ejecutándose en una máquina UNIX al mismo tiempo, por lo que es muy peligroso desconectar simplemente de la corriente el ordenador cuando te hayas cansado de utilizarlo (ya te veía las intenciones...). El sistema UNIX dispone de herramientas expresamente diseñadas para producir una secuencia ordenada de sucesos cuando desees desconectar la máquina. A esto se le conoce como procedimiento de desconexión, y se debería seguir muy cuidadosamente para asegurar el buen funcionamiento posterior del equipo y del sistema. También el procedimiento de arranque es complejo (o creías que sólo era apretar a ese botón tan majo???). En este apartado aprenderás los pasos que sigue el sistema cada vez que arranca y se desconecta, así como los problemas más comunes que pueden darse en los procedimientos de arranque.

3.1.- El Entorno del Sistema UNIX

Los demonios del sistema (no te asustes, no es ninguna peli de terror), están siempre ejecutándose en el sistema y posiblemente el administrador tenga un shell abierto y algunos programas funcionando bajo su sesión. Además, es muy probable que otros usuarios puedan estar conectados a la máquina desde terminales remotos y pueden estar ejecutando programas.

Además, la falta de sincronización entre los *buffers en memoria* y el disco duro del sistema significa que los contenidos “reales” y los contenidos “lógicos” del disco diferirán frecuentemente. Es decir, cuando escribes un fichero desde el editor, este fichero probablemente no se actualizará en el disco hasta segundos o minutos después de que se complete la operación de escritura y estés de vuelta en el shell ejecutando nuevas órdenes.

Todos estos factores hacen importante que tengas cuidado cuando desconectes el aparato. Cuando sea posible, utiliza las herramientas provistas para ayudar en estas tareas. Las versiones actuales de UNIX pueden soportar las caídas de tensión o potencia e, incluso, las desconexiones erróneas, aunque con algún riesgo de fallo para el sistema o para los contenidos del disco del sistema.

3.2.- Desconexión de la máquina

En principio, una desconexión correcta advertirá a los otros usuarios para que se despidan antes de que el sistema se apague (sí, ya sé que en tu casa nadie sabe informática nada más que tú, así que, Remigio, el único que lo usarás serás tú). Esta desconexión correcta eliminará cuidadosamente todos los procesos no esenciales, actualizará los diferentes ficheros y registros del sistema, sincronizará el disco con los buffers en memoria y, finalmente, eliminará el resto de los procesos.

La orden *shutdown*

Puedes usar varias herramientas diferentes para desconectar la máquina, y todas ellas son preferibles a que aprietes ese botoncito tan majo que pone: *POWER*. Una de estas herramientas es la más segura, aunque también la más lenta. Se trata de la orden **shutdown**, que es un guión localizado en */etc/shutdown*. El uso de esta herramienta sólo está reservada al superusuario (como Superman, pero en informático), es decir, al administrador del sistema. Sólo puede ser ejecutada en la consola del sistema, y solamente desde el directorio raíz (/). La orden **shutdown** protestará y rehusará actuar si estas condiciones no se cumplen (caprichosa que es ella...).

Los procedimientos de desconexión y los mensajes visualizados durante la desconexión varían entre las diferentes versiones del sistema UNIX (así que no busques ese menú tan bonito que tiene tu primo Dalmacio en el pueblo, porque seguramente tú no lo tengas).

La orden **shutdown** fue pensada originariamente para ser interactiva, con el usuario controlando las acciones tomadas durante el procedimiento de desconexión. Todavía puede ser utilizada interactivamente, pero revisiones recientes del UNIX permiten usar la opción **-y**, que instruye a shutdown a contestar a todas las cuestiones por sí misma.

Esa forma es mucho más fácil de usar que la orden sin la opción **-y**. Antes de ejecutar esta orden, es de cortesía verificar la actividad de los otros usuarios para asegurarnos de que no estén haciendo algo importante. Las órdenes **who** y **ps -af** determinarán la actividad actual del sistema. Nos ocuparemos de estas órdenes más adelante. Además, deberíamos verificar que ningún trabajo esté siendo impreso y que no haya transferencias de datos **uucp** en progreso, ya que estas operaciones volverán a comenzar desde el principio después de un arranque si son interrumpidas por la desconexión.

Cuando se ejecute, **shutdown** advertirá a todos los usuarios que la máquina va a ser desconectada pronto, y que deberían cerrar sus sesiones antes de que TÚ, como administrador (te gusta como suena eso, ¿verdad?), les cortes el suministro. La figura siguiente muestra una típica pantalla de consola durante una secuencia de desconexión.

```
# cd /
# shutdown -y

Shutdown started.      Mon Jun 15 20:29:49 EDT 1987

The system will be shut down in 60 seconds.
Please log off now.

THE SYSTEM IS BEING SHUT DOWN NOW ! ! !
Log off now or risk your files being damaged.

The system is coming down.  Please wait.

The system is down.
Reboot the system now.
```

Después de avisar a los usuarios mediante los mensajes de desconexión, la orden **shutdown** detendrá todos los procesos activos, actualizará el disco correctamente y paulatinamente llevará el sistema operativo hasta su parada. Finalmente, el sistema será suspendido hasta un punto en donde la alimentación puede ser desconectada o donde pueda iniciarse un nuevo arranque.

Por omisión, **shutdown** deja pasar 60 segundos entre el primero y el segundo mensaje de aviso de desconexión. Podremos alterar este intervalo añadiendo una opción **-g** a la línea de orden de **shutdown**. El número de segundos a esperar antes de comenzar efectivamente la secuencia de desconexión sigue a la opción **-g**. Por ejemplo, la orden

```
# shutdown -y -g300
```

hará que **shutdown** espere cinco minutos (así te tomas un café, eh listillo!!!) durante el período de aviso. Puedes alargar este período para permitir que los usuarios completen sus actividades antes de comenzar la desconexión o acortarlo para permitir una desconexión más rápida cuando sea segura. En casos extremos, se puede seleccionar **-g0** para reducir el intervalo de espera a cero. Esto puede ser peligroso si hay muchos terminales activos, ya que el mensaje de advertencia puede no visualizarse en todos los terminales antes de que el sistema se suspenda. A veces la desconexión fallará si no se completan todos los procesos de usuario antes de que el procedimiento de desconexión comience, y **-g15** es el intervalo seguro más corto en máquinas muy ocupadas.

3.3.- La secuencia de arranque

Cuando se aprieta ese botoncito que pone en marcha tu pedazo de sistema, la máquina sigue un complejo proceso de arranque. Esta secuencia puede tardar varios minutos, dependiendo del hardware y el software instalado en tu equipo, y no puede hacerse más rápida. El proceso de arranque incluye varios *cheques de sanidad*, y con frecuencia tratará de reparar cualquier daño encontrado, especialmente daños a ficheros en el disco rígido. La mayoría de máquinas UNIX tienen procedimientos incorporados que minimizan esta verificación de errores i la

desconexión anterior fue completada correctamente. Por tanto, la ausencia de arranque después de una caída de potencia o de alguna desconexión inadvertida será probablemente más compleja y completa que un arranque después de una desconexión normal. En cualquier caso, la secuencia de arranque suele ayudar a reparar los problemas del sistema

La tabla siguiente muestra una secuencia de arranque típica, según se visualiza en la consola del sistema.

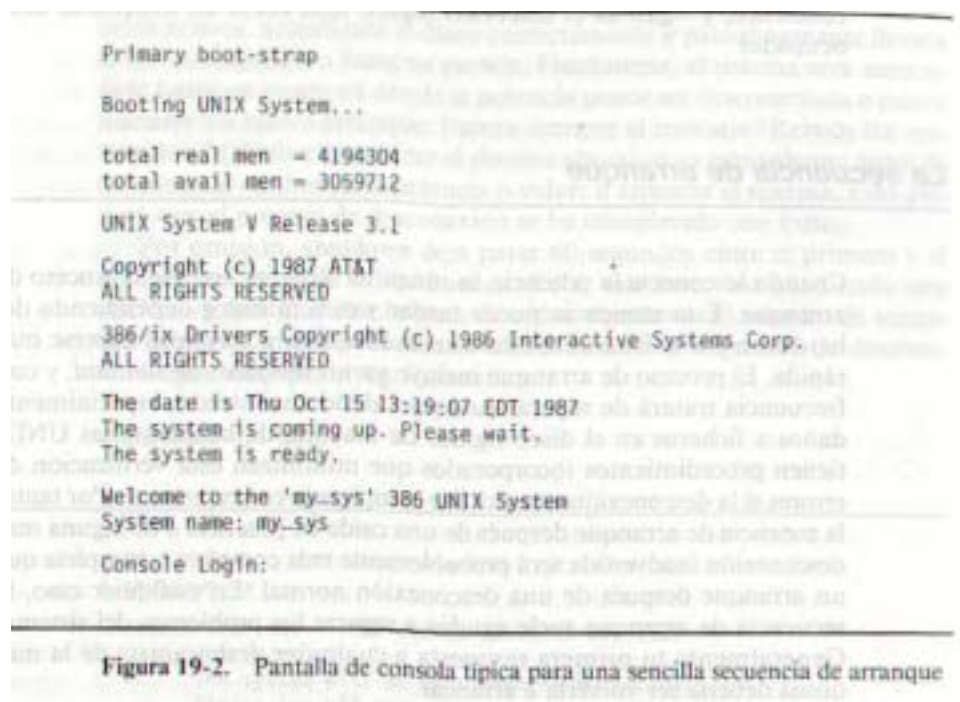


Figura 19-2. Pantalla de consola típica para una sencilla secuencia de arranque

Esta salida variará dependiendo del tipo de CPU en uso, la versión del sistema UNIX instalado, y el hardware o software adicional que haya en la máquina.

El primer mensaje “*Primary boot-strap*” es mostrado por el cargador ROM, que carga la primera parte del sistema operativo desde el disco. De hecho, la ROM cargará otro programa “cargador” que, a su vez, carga el propio sistema UNIX. Este “cargador software” adicional está almacenado en el disco del sistema, de modo que debe ser cargado por módulos hardware y ROM que existen permanentemente. Diferentes sistemas operativos guardan estos programas cargadores en la misma ubicación relativa del disco dentro de sus particiones de disco con objeto de que el cargador ROM pueda hallar el cargador software. Esto permite conmutar entre sistemas operativos cambiando simplemente la partición implícita desde la cual la ROM carga el cargador software.

El cargador software es traído a la memoria, y la ROM le cede el control, comenzando así su ejecución. La máquina está ahora obligada a ejecutar el sistema

UNIX, ya que el cargador software sólo puede tratar con su propio sistema operativo. Cuando el cargador software comienza, muestra el mensaje,

Booting UNIX System...

Y carga entonces el *núcleo* del sistema operativo, que normalmente es **/UNIX**. Podemos presionar una tecla en el teclado mientras se visualiza el inductor “Booting...”, y el cargador te permitirá introducir el nombre de camino de un núcleo alternativo a cargar.

En otras revisiones, el mensaje será el siguiente:

Boot:

Podemos escribir el nombre de un núcleo alternativo si lo deseamos. Normalmente estaremos quietecitos y después de un breve intervalo el cargador mostrará el núcleo por omisión. Si queremos cambiar el núcleo podemos presionar el <TAB>. Si no hemos seleccionado ningún núcleo alternativo, se comenzará a cargar el sistema UNIX.

Este fichero, **/UNIX**, es visible en el disco rígido en el directorio raíz. Es el núcleo, la porción efectiva residente en memoria del sistema UNIX. El cargador software lo leerá desde disco y lo instalará en la memoria de la máquina. El cargador software cederá luego el control al núcleo recién cargado, y el sistema UNIX comenzará a inicializarse a sí mismo.

Como parte de la secuencia de inicialización, el núcleo puede mostrar su idea de cuánta memoria real hay instalada en el sistema. Versiones más antiguas requieren con frecuencia la compilación de la información de memoria en el sistema como el **uname** implícito. Si se añade más memoria al sistema, el mensaje de la inicialización cambiará, y el sistema UNIX se adaptará a sí mismo correctamente. Si el mensaje de “Total real memory” difiere de la memoria física que sabemos que tiene nuestro equipo, deberíamos reparar el error antes de volver a usar la máquina. La “Available memory” muestra la memoria que el sistema puede poner a disposición de los procesos normales después de que el sistema UNIX toma la que necesita. Parecería que la diferencia entre estos dos números debería ser la memoria real utilizada por el núcleo. Pero de hecho este número es mucho mayor que el tamaño del fichero **/UNIX** según podemos averiguar con **ls -l**. La diferencia se debe a que el sistema UNIX asigna cantidades significativas de memoria a sus *buffers*, que actúan en cierto modo como un disco RAM.

Los programas normales puede utilizar la *memoria disponible* cuando se ejecuten. Sin embargo, los sistemas UNIX modernos pueden intercambiar o paginar segmentos de memoria desde RAM a disco cuando el sistema necesita más memoria. Por tanto, puede haber ejecutándose muchos más programas de lo que se podría esperar a partir del mensaje “Available memory”. Como es habitual, mientras más memoria real haya disponible, más eficaz puede ser el sistema UNIX, debido a que se reduce el intercambio con disco.

Después, el sistema UNIX comienza a inicializarse a sí mismo y a los dispositivos hardware instalados. Cuando se completa la inicialización, el sistema UNIX está realmente activo y corriendo, aunque aún deben hacerse muchas más tareas antes de que el sistema esté listo para mostrar la pantallita de inicio. Hasta que aparece la pantalla del inductor “login” se produce la activación de los guiones shell de tiempo de arranque.

En un sistema operativo correcto no se requiere ninguna acción entre la conexión de la potencia y la eventual aparición del indicativo “login”. Sin embargo, si la máquina sufre una caída justo antes de que la vuelvas a arrancar, con frecuencia aparecerá un inductor adicional en la secuencia de arranque.

There may be a system dump memory image in the swap device

Do you want to save it (y/n)?

Este volcado o imagen de memoria es para depurar el núcleo. Presiona **N** para eludir este paso.... ¿ESTÁS SEGURO?

3.4.- Los Estados INIT

El procedimiento de arranque del sistema UNIX se complica por la posibilidad de hacer entrar el sistema en diferentes estados. Es decir, el sistema puede adoptar varios modos diferentes de operación. Éstos son conocidos como los diferentes *estados init*, debido a **/etc/init**, que es el programa responsable de mantener el sistema funcionando correctamente. Estos estados son muy diferentes unos de otros, y el sistema sólo puede estar en uno de ellos en cualquier instante. Los procedimientos de desconexión y arranque controlan realmente el estado en que se encuentra la máquina-

El estado más comúnmente utilizado es el llamado modo multiusuario. Este estado del sistema se utiliza para casi todas las interacciones y es el único que permite más de un usuario. Otro estado que fue históricamente importante, pero que ahora apenas se usa, es el llamado modo de usuario único. Este estado es una versión multitarea del sistema UNIX, de modo que permite múltiples procesos, pero no es multiusuario. Es decir, solamente la consola del sistema está activa cuando la máquina opera en modo de usuario único. Generalmente es un error si el sistema entra en el modo de *usuario único*

Hay otros estados además de los modos de usuario único y multiusuario. Todos ellos se designan mediante identificaciones especiales, como se muestra en la tabla siguiente:

Estado	Función
0	Desconectar la máquina
1	Modo de usuario único
2	Modo multiusuario

3	Multiusuario con RFS
4	No utilizado
5	Desconexión y arranque (no usado)
6	Desconexión a ROM (no usado)
s	Modo de usuario único
S	Modo de usuario único con consola remota

El estado **init 0** se usa para suspender la máquina y detener el sistema UNIX. El modo de usuario único es conocido como estado **init 1**, **init s** o **init S**, dependiendo de si el terminal activo único es la consola del sistema o un terminal remoto. El modo multiusuario es conocido como estado **init 2** y el **init 3** se usa en los sistemas SVR3 cuando está activa la capacidad RFS (Remote File System). El estado **init 4** casi nunca se utiliza, pero los estados **5** y **6** se emplean en algunas máquinas para significar “desconexión y arranque”, y “desconexión para el arranque desde el ROM”, respectivamente. La mayoría de las máquinas SVR3 no utilizarán los estados **4**, **5** ó **6**.

Cambio del estado init

La orden **shutdown** es útil para algo más que para desconectar la máquina: realmente está diseñada para cambiar el estado **init**. Por omisión, **shutdown** llevará la máquina al estado **0**, preparándola así para desconectar la potencia del sistema. Sin embargo, el argumento **-i** sirve para especificar explícitamente el estado **init**.

```
# shutdown -y -g45 -i0
```

Esto suspenderá el sistema, mientras que

```
# shutdown -y -i1
```

llevará la máquina al estado de usuario único. Normalmente el sistema se encontrará en el estado **2**, a menos que se esté utilizando las facilidades RFS, en cuyo caso estará en el estado **3**.

Estos estados son importantes para comprender el procedimiento de arranque, ya que los guiones shell que se ejecutan durante todo el procedimiento de arranque diferirán dependiendo de a qué estado se dirija el sistema.

3.5.- El Fichero /etc/inittab

Tras la determinación de la inicialización interna, el sistema comenzará el demonio **/etc/init**, el cual asume el control de la secuencia de arranque. El proceso **init**, que permanecerá activo durante todo el tiempo que el sistema esté corriendo, sirve una función muy importante: se asegura de que todos los demás demonios del sistema se ejecuten cuando deben. Por ejemplo, cuando te presentas

ante la máquina, tu shell reemplaza al programa **getty** de modo que **getty** ya no existe para el terminal. Sin embargo, cuando te despidas de la máquina, tu shell morirá, dejando un puerto de terminal muerto que no puede ser utilizado. El proceso **init** debe reconocer cuándo muere tu shell y *reengendrar* el **getty** para tu puerto terminal. Esto da lugar a una nueva visualización del inductor “login:” en el terminal. Realmente **init** es informado de la muerte del shell a través de una señal del sistema, y toma entonces la acción apropiada. De hecho, **init** tiene este efecto también sobre otros varios demonios del sistema, y generalmente asegura que los procesos importantes del sistema estén ejecutándose. Naturalmente, si el propio **init** muere, no habrá ningún proceso que lo regenere, y el sistema gradualmente irá descomponiéndose más y más hasta que caiga definitivamente. Afortunadamente, es muy difícil eliminar a **init**.

El proceso **init** obtiene sus instrucciones del fichero **/etc/inittab**. Los contenidos de este fichero controlan todos los estados **init**, y el fichero controla también qué procesos deben ser regenerados cuando mueran. El fichero **/etc/inittab** es una base de datos típica del sistema UNIX, con líneas formadas por varios campos separados por (:). La siguiente figura muestra un **inittab** típico de un sistema SVR3 pequeño.

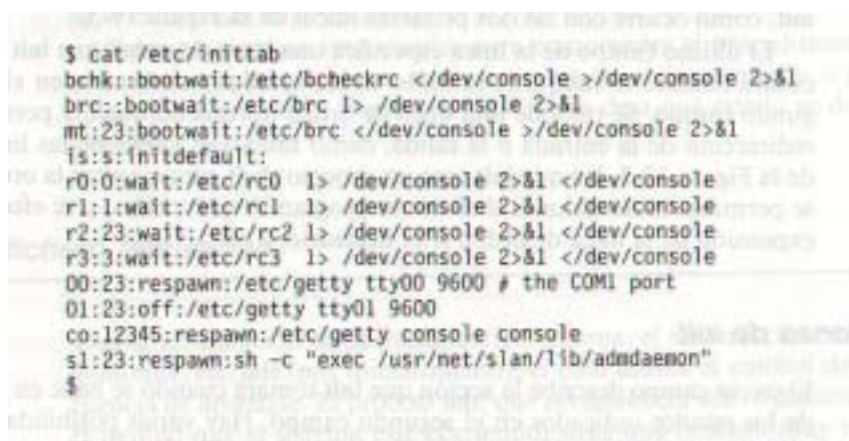


Figura 19-3. Fichero típico **/etc/inittab** para un sistema SVR3 pequeño

Sus contenidos diferirán en gran medida dependiendo del software instalado en la máquina, el número de terminales remotos permitidos y la versión del propio sistema UNIX. Cuando **init** comienza, lee cada línea del fichero y toma una acción dependiendo del contenido de la línea.

El primer campo de cada línea es un identificador que “designa” efectivamente a cada línea, y que debería ser único. El segundo campo define los estados **init** para los cuales la línea está activa. Puede contener más de un estado, como en **23**, que define la línea activa en los estados **init 2** y **3**. Si este campo está vacío, la línea estará activa en todos los estados **init**, como ocurre con las dos primeras líneas de la figura adjunta.

El último campo de la línea especifica una línea de orden que **init** ejecutará cuando la máquina se halle en los estados nombrados en el segundo campo. Se trata de una línea de orden normal del shell, y permite redirección de la entrada o la salida, como muestran varias de las líneas de la figura anterior. Ya que **init** crea un proceso shell para ejecutar la orden, se permiten tanto guiones shell como programas ejecutables, y se efectúa expansión de la línea de orden si es necesario.

Acciones de *init*

El tercer campo describe la acción que **init** tomará cuando se halle en uno de los estados indicados en el segundo campo. Hay varias posibilidades: **off** instruye a **init** a eliminar la orden nombrada si existe, mientras que **once** instruye a **init** a ejecutar el programa cuando entre al estado indicado pero si esperar su terminación. El proceso **init** continuará con su trabajo, sin advertir si la orden acaba o continúa ejecutándose. La orden **wait** hace que **init** ejecute el programa cuando entre a los estados indicados, pero esperando hasta que el proceso se complete antes de continuar. Por tanto, puedes ejecutar varias órdenes en una secuencia específica a través de **init**, ya que el operador **wait** hace que la orden de una línea finalice antes de que se ejecute la siguiente línea. Las órdenes que especifiques con **boot** y **bootwait** se ejecutarán únicamente cuando **init** lea el fichero **inittab** en tiempo de arranque (y no cuando lea el fichero en otras ocasiones). Estas órdenes se diferencian en que **bootwait** hace que **init** espere a que la orden se complete, mientras que **boot** no. La orden **initdefault** tiene un significado especial, y no incluirá un campo de orden. Instruye a **init** a entrar al estado indicado cuando se inicie por primera vez. Finalmente, **respawn** instruye a **init** a iniciar el proceso cuando entre a los estados indicados y a reiniciar el programa cada vez que **init** detecte que el programa ya no está ejecutándose. Otras varias órdenes son posibles en el segundo campo, pero son raramente utilizadas en la mayoría de las máquinas.

Procesamiento en tiempo de arranque

Con esta información, ya podemos acceder al fichero **/etc/inittab** e intentar comprender lo que hace el sistema cuando lee este fichero. En el ejemplo anterior, la primera línea (**bchk**) comienza la secuencia de arranque ejecutando el guión contenido en **/etc/bcheckrc**. Esto se aplica en todos los estados **init** y se ejecuta únicamente en el tiempo de arranque del sistema. La orden **bchk** verifica el sistema de ficheros antes de dejar que el sistema operativo trabaje con él. Debido a una comprobación preliminar para determinar si el sistema de ficheros es tenido por correcto, este guión no tendrá mucho efecto cuando el sistema haya sido desconectado correctamente. Sin embargo, si el sistema se suspendió debido a una caída de tensión o a causa de algún error interno, la orden **fsck** será ejecutada para arreglar las cosas.

La segunda línea (**brc**) también se ejecuta en tiempo de arranque. No se ejecuta hasta después de completarse **bchk**, y el sistema no proseguirá hasta que ésta se complete. La orden **brc** es responsable de inicializar los sistemas de ficheros *activos* en la máquina. Esto incluye los sistemas de ficheros estándar, normalmente **/root** y **/usr**, además de los sistemas de ficheros remotos disponibles bajo la

facilidad RFS. La orden **brc** se ejecuta de nuevo en la secuencia de arranque desde la línea **mt** si el sistema se dirige a los estados **init 2** ó **3**, los modos multiusuario.

La siguiente línea (**is**) trata de decir a **init** que entre al modo de usuario único cuando comience (en tiempo de arranque). Si no existe una línea **initdefault**, **init** solicitará desde la consola del sistema un estado en el que entrar. Puesto que esto generalmente es indeseable, la entrada **initdefault** está casi siempre presente. Hay un ligero truco aquí, ya que **init** rehusará entrar al estado **1** desde una entrada **initdefault**. Por tanto, esta línea tiene el único efecto de suprimir la petición del *nivel de ejecución* que **init** de otra manera efectuaría en tiempo de arranque. Luego **init** se dirige al estado **2** (modo multiusuario).

Guiones rc

Las líneas siguientes especifican los guiones a ejecutar cuando se soliciten estados específicos. Todos ellos son guiones shell. Generalmente se ejecutará **/etc/rc2**, ya que el estado **2** es el utilizado con más frecuencia. De nuevo, **init** esperará a que el guión se complete antes de continuar. El guión **/etc/rc2** contiene gran parte del código de inicialización específico para crear el entorno de operación normal. Los contenidos de este guión difieren en las diferentes versiones, pero en los sistemas SVR3 su función primaria es examinar el directorio **/etc/rc2.d**.

```
$ ls -C /etc/rc2.d
```

K30fmounts*	S01MOUNTSFSYS*	S21perf	S70uucp*
K40rmounts*	S05RMTMPFILES*	S22acct	S75cron*
K65rfs*	S20syssetup*	S67starlan	xK67nls*

S

Posiblemente, el contenido de este directorio sea distinto dependiendo de cada sistema. El guión **/etc/rc2.d** ejecutará los guiones de ese directorio en una secuencia específica: todos los ficheros cuyos nombres comiencen con **K** se ejecutan primero, en secuencia ordenada por sus nombres. Luego se ejecutan todos los ficheros con nombres que empiecen por **S**, también en orden. Este procedimiento permite a los programadores activar nuevas funciones durante los cambios de estado de **init** simplemente añadiendo nuevos ficheros al directorio **/etc/rc2.d**.

En el ejemplo anterior, los guiones **K** desmontarán cualesquiera recursos compartidos creados por la capacidad **RFS**. Esto es necesario, ya que podríamos haber estado en el estado **3** previamente. Los guiones **S** vuelven a montar los sistemas de ficheros locales, limpian de ficheros los directorios temporales del sistema, inician la contabilidad de procesos y usuarios del sistema, limpian los directorios de **uucp**, prepara la red de área local e inician la ejecución del programa **cron**. Estas funciones son generalmente más complejas que simples líneas de orden en el **inittab**, y se les da el estatus de guiones shell completos en lugar de simples líneas con la acción **bootwait** en el fichero **inittab**. Únicamente se ejecutan los ficheros que comiencen por **K** o **S**, de modo que el fichero **xK67nls** es ignorado

por **/etc/rc2**. Los números de los nombres de ficheros proporcionan información de secuencia para que **/etc/rc2** sepa el orden en que ejecutar los guiones.

Este mismo esquema se aplica cuando **init** entra a los estados **0**, **1** y **3**. El guión **/etc/rc0**, **/etc/rc1** y **/etc/rc3** se ejecuta en el caso apropiado, el cual a su vez explora el directorio **/etc/rc0.d**, **/etc/rc1.d** o **/etc/rc3.d**, según corresponda. Siguiendo la secuencia de estos guiones y los contenidos de los directorios, se puede rastrear la secuencia completa de acciones que el sistema UNIX atravesará cuando cambie de estado.

Procesos *getty*

Cuando se completen estos guiones, **init** continuará explorando el fichero **inittab**. Generalmente lo único que queda es definir los procesos demonio permanentes que se regeneran si son eliminados por alguna razón. Las últimas cuatro líneas de la figura anterior son de este tipo, aunque puede haber muchas de ellas en una instalación grande. Probablemente serán procesos **getty** que atienden a un puerto de comunicaciones inactivo a la espera de que se produzca una presentación, desde un terminal o un MODEM. Se requiere un proceso **getty** por cada puerto de comunicación en el que se permita la presentación de un usuario. Cuando un usuario se conecte a ese puerto, **getty** desaparecerá, aunque **init** no lo tratará como eliminado hasta que el usuario se despida del puerto y cuelgue el terminal. Entonces **init** regenerará el proceso. Las líneas 00 y 01 de la figura anterior son ejemplos de entradas **getty**, aunque sólo 00 está activa; 01 está puesta a **off** de modo que el puerto COM2 está discapacitado. COM1. Permitirá presentaciones desde un terminal asociado o desde un MODEM. Estos puertos terminales solamente están activos en modo multiusuario. La línea **co - getty** para la consola del sistema – es necesaria para proporcionar acceso superusuario a la máquina, por lo que está activa para todos los estados **init**. La última línea, **sl**, es una línea especial requerida por el software de red de área local.

Realización de cambios en el fichero *inittab*

Con frecuencia deberemos efectuar cambios en el fichero **inittab** cuando cambiemos la configuración del sistema. Por ejemplo, para desactivar el **getty** para un puerto remoto, cambiaremos la acción **respawn** por **off**, o cambiaremos la acción **off** por **respawn** para iniciarlo. Además, a menudo cambiaremos la velocidad implícita de la línea **getty** cuando cambiemos los dispositivos conectados a un puerto. Estos cambios deben ser realizados por el superusuario, que puede editar el fichero **inittab** cuando lo desee.

En revisiones SVR3 más recientes, los cambios a **/etc/inittab** no sobrevivirán a la adición de paquetes hardware/software que requieran reconfigurar el núcleo para instalar nuevas rutinas de dispositivo. Después de añadir nuevos paquetes, deberíamos verificar si **/etc/inittab** ha retenido los cambios que hayamos hecho, y añadirlos de nuevo si se han perdido.

El programa **init** lee el fichero **inittab** solamente una vez, cuando se inicia. Si efectuamos cambios al fichero, no tendrán efecto hasta que informemos a **init** que el fichero ha cambiado. Usaremos el programa **telinit** para esto

El argumento **q** de **telinit** instruye a **init** a volver a leer el fichero **inittab** y tomar acciones basadas en los cambios habidos en el fichero desde la última vez que **init** lo examinó. No se produce cambio en el estado **init**. El programa **telinit** está reservado al superusuario.

Además, podemos usar la orden **telinit** para modificar el estado **init**. Esto se efectúa raramente en los sistemas modernos, y debería ser realizado con cuidado. Además de **q**, **telinit** puede tomar un argumento para indicar un nuevo estado **init**. Así, la orden siguiente

```
$ telinit 1
```

llevará al sistema al modo de usuario único, mientras que la orden

```
$ telinit 0
```

suspenderá el sistema para desconectarlo. Pero este uso de **telinit** no es recomendable. Desde el modo de usuario único deberíamos suspender la máquina completamente y volverla a arrancar para cambiar a modo multiusuario.

Un procedimiento de desconexión más corto

Raramente deberíamos tratar de acelerar el procedimiento de arranque, aunque pudiéramos desear que tardara menos tiempo. Ciertamente, considerando las muchas tareas complejas que se realizan en tiempo de arranque, podemos tolerar la relativamente larga secuencia de arranque. Sin embargo, sí podemos acelerar el proceso de desconexión. Con frecuencia no hay usuarios conectados a la máquina desde terminales ajenos. Si sabemos que la máquina está en un estado de relativa quietud, podemos evitar la orden **shutdown** y suspender la máquina muy rápidamente. Tres acciones diferentes son necesarias. Primero, los buffers del sistema UNIX deben sincronizarse con el disco para que el disco quede actualizado. Segundo, deberíamos desmontar cualesquiera sistemas de ficheros adicionales al sistema de ficheros raíz. Tercero, el marcador de sanidad del disco debe ser correcto, de modo que no sea necesaria la comprobación del sistema de ficheros cuando volvamos a iniciar la máquina. Este procedimiento sólo está disponible en los sistemas SVR2 y SVR3, y no en versiones anteriores.

La primera exigencia se gestiona con la orden **sync**, que actualiza el disco rígido de la máquina para que el disco sea correcto. Esta orden se ejecuta normalmente dos o tres veces por sesión.

```
# sync
```

```
# sync
```

```
#
```

La orden **sync** está reservada al superusuario.

Segundo, podemos desmontar los sistemas de ficheros montados adicionalmente al sistema de ficheros raíz con las órdenes **umount** o **umountall**. Esto retira con seguridad los sistemas de ficheros adicionales.

La tercera acción se gestiona con la orden **uadmin**. Esta orden toma un argumento que especifica la acción a realizar, y utiliza el argumento para desconectar la máquina inmediatamente, después de marcar el disco como sano.

```
# uadmin 2
```

Reboot the system now

Cuando aparezca el mensaje “*reboot*”, podemos desconectar o volver a arrancar la máquina. Esto generalmente lleva tan sólo de 2 a 3 segundos. El argumento **2** no es un estado **init**, sino un código especial utilizado únicamente por **uadmin**. Debemos utilizar esta secuencia de desconexión con cuidado, y solamente cuando sepamos que la actividad del sistema es extremadamente baja, sin usuarios, sin trabajos de impresión y sin transferencias de datos remotos en proceso.

La orden *fsck*

Otra acción importante más asociada generalmente con el procedimiento de arranque es la *verificación del sistema de ficheros*. Puesto que el sistema UNIX depende tanto de la sanidad del sistema de ficheros, existe una herramienta especial para verificarlo y repararlo. Se trata de la orden **/etc/fsck**, y que está reservada al superusuario.

Sus funciones son muchas y su uso es complejo. En tiempo de arranque, el guión **/etc/bcheckrc** comprueba si el indicador de sanidad del sistema de ficheros fue escrito como parte de la secuencia de desconexión. La orden **shutdown** escribirá este indicador correctamente, pero las desconexiones inesperadas no lo harán. Cuando el indicador exista, el procedimiento de arranque supondrá que el sistema de ficheros es correcto (incluso si no lo es). La orden **fsck** no será ejecutada y la secuencia de arranque será notablemente más rápida. Si el indicador no está establecido correctamente, el guión **bcheckrc** supondrá que el sistema de ficheros puede haber sido *dañado* y ejecutará automáticamente **fsck**. Cuando se ejecute **fsck** aparecerá salida adicional en pantalla de consola en tiempo de arranque, como se muestra a continuación.

```
# fsck /dev/rdisk/0s1
/dev/rdisk/0s1
File System: rootfs Volume: disk0
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
UNREF FILE I=2680 OWNER=listen MODE=20000
SIZE=0 MTIME=Jun 22 08:22 1987
CLEAR? y
UNREF FILE I=3540 OWNER=listen MODE=20000
SIZE=0 MTIME=Jun 22 08:22 1987
CLEAR? y
UNREF FILE I=3541 OWNER=listen MODE=20000
SIZE=0 MTIME=Jun 22 08:22 1987
CLEAR? y
FREE INODE COUNT WRONG IN SUPERBLK
FIX? y
** Phase 5 - Check Free List
SET FILE SYSTEM STATE TO OKAY? y
3560 files 61968 blocks 40310 free
*** FILE SYSTEM WAS MODIFIED ***
#
```

Figura 19-4. Salida típica de fsck

La orden **fsck** también puede ser ejecutada en la consola por el superusuario, con un nombre de sistema de ficheros a verificar como argumento, (como en la figura anterior), pero puede producir gran daño al sistema de ficheros en manos no experimentadas. Deberíamos volver a arrancar la máquina cuando deseemos una verificación del sistema de ficheros a menos que sepamos *exactamente* cómo usar **fsck**. Cuando usamos **fsck** manualmente, lo ejecutamos sobre un *sistema de ficheros en bruto, sin montar*, en lugar de sobre un sistema de ficheros activamente montado.

La orden **fsck** ejecuta cinco fases diferentes durante la verificación del sistema de ficheros. Primero, comprueba las tablas internas de tamaño de los ficheros frente al tamaño real de los ficheros en el disco. Segundo, verifica la sanidad de los nombres de camino a directorios y ficheros. A continuación, verifica la corrección de la conectividad entre los ficheros y sus directorios padres. Cuarto, verifica el número de vínculos entre los ficheros y sus nombres para asegurarse de que los ficheros estén correctamente referenciados. Por último, verifica que todos los bloques del disco no utilizados estén correctamente incluidos en la *lista de bloques libres* del sistema de ficheros. Cuando aparecen errores, **fsck** muestra un mensaje de error o solicita al superusuario que disponga del fichero de alguna manera, ya sea suprimiéndolo o revinculándolo en el sistema de ficheros. Las respuestas *y* a las cuestiones mostradas en la figura inmediatamente anterior son ejemplos de respuestas manuales a estas peticiones. Cuando se ejecuta **fsck** como parte del

procedimiento **fsck** en tiempo de arranque se ejecuta sobre cada sistema de ficheros que estuviera en uso cuando el sistema fue suspendido.

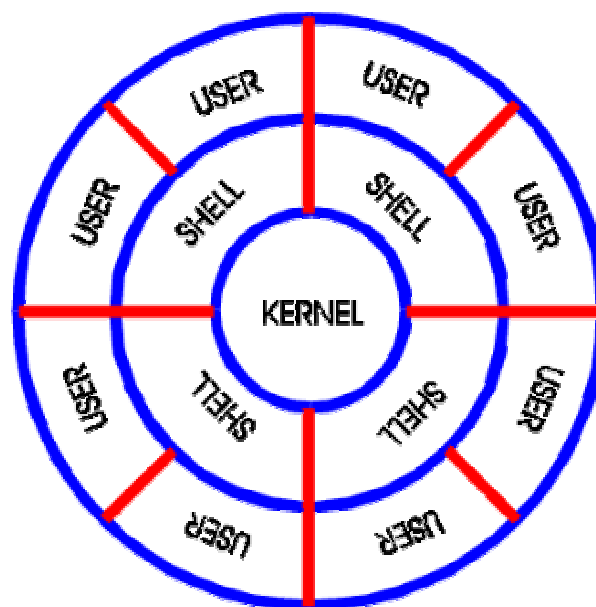
Cuando **fsck** encuentra un fichero o parte de un fichero que no está correctamente vinculado en el sistema de ficheros, lo revincula al sistema en un lugar especial. Se trata del directorio **lost + found**, y uno de tales directorios debería aparecer en el directorio raíz de cada sistemas de ficheros montado. Estos serán generalmente **/lost + found** y **/usr/lost + found** si la máquina tiene dos sistemas de ficheros. Después de completarse el arranque, el superusuario puede inspeccionar estos ficheros o partes de ficheros para asegurarse de que no haya nada de valor allí. Los ficheros que haya en esos directorios fueron colocados allí al encontrar **fsck** que contenían errores, por tanto, deberíamos examinar los ficheros que aparezcan en los directorios **lost + found** ya que provienen de algún otro lugar del sistema de ficheros. A veces son vínculos extraviados sin contenido útil, pero podrían ser ficheros críticos que estuvieran siendo actualizados por el sistema cuando la máquina se suspendió. Trataremos a los ficheros de **lost + found** muy cuidadosamente; siempre es difícil determinar lo que significan y de dónde provienen en el sistema de ficheros. Sin embargo, los directorios **lost + found** pueden llegar a crecer demasiado, por lo que deberías inspeccionarlos ocasionalmente y suprimir los ficheros desconocidos.

El *Kernel* y los *Shells*

En realidad el kernel es la gran componente del sistema operativo UNIX. El archivo asociado con él se encuentra siempre en el directorio raíz (root). Cuando el sistema arranca este archivo (programa) se ejecuta y permanece activo hasta que el sistema se descarga o apaga.

El kernel interactúa entre los dispositivos del equipo (hardware) y los interpretadores (shells) del sistema. Todos los dispositivos entienden lenguaje de máquina y microprogramación. Al darle al kernel la libertad de interacción, hace que el usuario no tenga que aprender lenguaje de máquina y le permite trabajar más eficientemente a otro nivel.

Cuando el sistema operativo UNIX se instala en una máquina, el kernel se construye para esa máquina en particular. Al hacerlo toma en cuenta todos los dispositivos y características del equipo y genera un archivo diseñado para interactuar óptimamente con esa máquina. Cualquier cambio considerable (como cambiar el número de inodos u otros parámetros), requiere que se tenga que generar otro kernel y que la computadora se apague y vuelva a arrancar para que los cambios surtan efecto. Casi nunca es necesario generar un nuevo kernel después del que se obtiene en la instalación. La siguiente figura ilustra como el usuario está totalmente aislado del kernel por el shell.



Algo importante que no puede pasarse por alto es que el kernel es la única parte del sistema operativo que se carga en memoria cuando usted enciende el equipo y permanece así hasta que se apaga. Cualquier otro proceso se carga o llama cuando se necesita y se detiene o descarga cuando ya no se requiere.

4.1.- Funciones del KERNEL

Además de interactuar con el hardware (equipo), el kernel es responsable del manejo de memoria, ejecución de trabajos y administración del sistema de archivos (file system). En términos generales el kernel hace lo siguiente:

- Asigna memoria a programas.
- Maneja las solicitudes de entrada y salida.
- Reparte el tiempo de procesamiento y lo comparte entre los programas.

Para ayudar al kernel a monitorear el sistema y ejecutar trabajos, UNIX incorpora los *daemons*.

Los daemons son procesos del kernel que están siempre activos y en alerta de procesos que necesitan atención. Los daemons existen para servicios de impresión, ejecución automática de trabajos (cron) y otros.

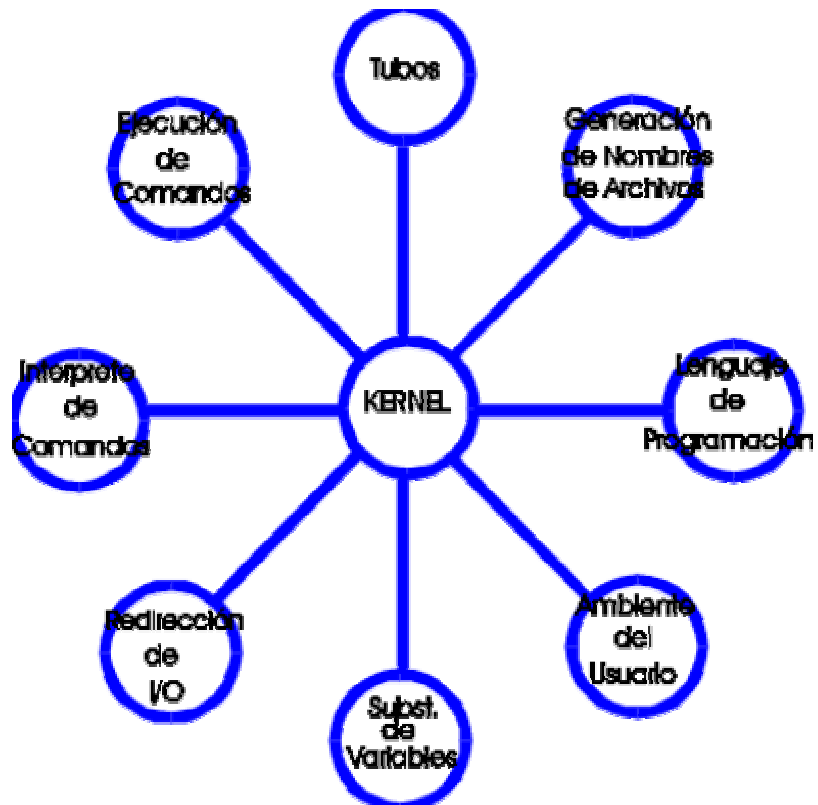
4.2.- El Shell

El *Shell* es el software que atiende a las órdenes tecleadas en el terminal y las traduce a instrucciones en la sintaxis interna del sistema. El nombre *Shell* (caparazón) describe realmente su función: es un material duro que se halla entre el núcleo del sistema y el mundo exterior, proporcionando una robusta interfaz de usuario para el sistema operativo. El shell incluye inusualmente un gran número de funciones; de hecho, implementa alguno de los conceptos más potentes y elegantes del sistema UNIX. Se han dedicado muchos esfuerzos a la mejora del shell y a la creación de nuevas versiones, para propósitos especiales. El shell es tan vital que generalmente no se considera aparte del sistema como un todo. Muchas capacidades y funciones provienen de servicios proporcionados por el shell.

El shell de UNIX es afortunadamente algo más que un intérprete de comandos u órdenes. Entre sus funciones destacan las siguientes:

- Interpreta órdenes y comandos.
- Ejecuta órdenes.
- Configura el entorno.
- Asigna valores a las variables.
- Sustituye variables por su valor.

- Genera nombres de archivo.
- Redirecciona entradas y salidas.
- Encauza listas de órdenes o tuberías.
- Sustituye elementos en la línea de comandos.
- Es un completo lenguaje-intérprete de programación.



4.3.- Tipos de SHELL

Para cada usuario se genera una copia de Shell en cada sesión. El tipo inicial de shell de presentación lo asigna el administrador, y figura en la línea o entrada correspondiente al usuario en el archivo `/etc/passwd`.

Hay varios tipos de shell disponibles en UNIX, y distintos usuarios pueden utilizar a su gusto diferentes shell a la vez en un mismo sistema, además de nuevos “sub-shells”.

El shell estándar es conocido como *shell Bourne*, en honor de su creador, Steve Bourne. Este pequeño shell fue diseñado para uso de propósito general y es relativamente eficaz. Con los años (el shell Bourne fue introducido alrededor de 1978), más características y mejoras, tales como las funciones shell, han sido añadidas para mantenerlo actualizado con el resto del sistema UNIX. En primer

lugar, no hay *historia de órdenes* (no hay manera de repetir una línea de orden sin reescribirla manualmente). Además, el shell estándar carece de soporte de *apodos* (alias) — la capacidad de los usuarios de adaptar los nombres de las órdenes que más utilizan —. Se puede resolver estos dos problemas del shell Bourne creando nuevas órdenes como programas shell, salvándolas en un fichero, y dando al fichero el nombre que deseemos. Pero esta solución es ineficaz, ya que se requiere un subshell para leer el programa y ejecutar sus contenidos. Finalmente, no hay modo rápido y fácil de editar las órdenes al vuelo, ya que el programa shell está relativamente permanente en el fichero.

Estas consideraciones motivaron el desarrollo de dos shells mejorados populares, el *Shell C* y el *Shell Korn*. Ambos shells rempazan al shell Bourne en aquellas áreas discutidas anteriormente. El shell C fue originalmente desarrollado como parte de la versión BSD, y el shell Korn fue desarrollado por David Korn de los laboratorios Bell AT&T en respuesta al shell C. Aunque ninguno de estos dos shells es parte estándar del sistema SVR3, ambos están plenamente disponibles como paquetes de software adicionales.

El Shell C fue diseñado únicamente para uso interactivo. Es decir, no puede ser nominado **/bin/sh** y por tanto no puede ser utilizado para todas las cosas que el sistema hace con el shell a espaldas del usuario. El shell c también es relativamente ineficaz comparado con el shell Bourne o el shell Korn. No obstante, tiene muchos devotos, especialmente entre los entusiastas de los sistemas UNIX BSD.

El shell Korn tiene características para uso interactivo y también puede remplazar completamente al shell Bourne como **/bin/sh**. Es mayor que el shell Bourne, pero es notablemente más eficaz, ya que dispone de más funciones internas que pueden ser efectuadas directamente por el shell y no requieren un subshell aparte.

Ambos shell recientes proporcionan muchas mejoras respecto a **/bin/sh**, incluyendo más operadores de programación shell, operadores aritméticos internos para remplazar a la orden **exp.** y mejores características de manejo de cadenas. Pero las ventajas más importantes de los shells Korn y C son: *edición de órdenes*, *historia de órdenes* y *apodos*.

La edición de órdenes se refiere a la capacidad de utilizar órdenes de edición normales de **vi** o **emacs** para modificar líneas de órdenes mientras están siendo introducidas. Las variables de entorno te permiten definir qué conjunto de órdenes de edición prefieres. Luego puedes utilizar estas órdenes de edición familiares para alterar el comportamiento de la entrada de línea normal y pasar a modo de edición. Por ejemplo, si tecleas una línea de órdenes larga que incluya un error con el modo de edición **vi** activo, puedes presionar <ESC> para entrar al modo de edición y luego moverte por la línea de órdenes con las órdenes de movimiento de cursor normales de **vi** tales como **b**, **3w**, **x**, etc. Están permitidas la búsqueda y sustitución con expresiones regulares. Cuando la línea de órdenes esté correcta, presiona <NEWLINE> y el shell ejecutará la orden resultante.

La historia de órdenes se refiere a la capacidad del shell de mantener un registro de todas las órdenes que introduzcas durante tu sesión. La historia se utiliza junto con la edición de órdenes para recordar una orden anterior que hayas ejecutado y ejecutarla de nuevo sin volverla a teclear o después de haberla editado. Estas herramientas son muy útiles cuando estás depurando líneas de cauce complejas o guiones shell, ya que puedes efectuar modificaciones rápidamente en órdenes relativamente largas sin invocar a un editor normal o teclear de nuevo la orden completamente.

La otra característica principal de los shell Korn y C es el apodo de órdenes. Un apodo es un nombre de orden definido para el shell con un operador **alias** especial, que define el nombre que deseas utilizar e incluye una definición de una orden real a sustituir cuando el apodo sea visto al comienzo de la línea de orden. Por ejemplo, puedes definir lo siguiente:

```
$ alias ls="ls -FC"
```

```
$
```

Esto define un apodo para la orden **ls**, de modo que cada vez que introduzcas

```
$ ls
```

la orden que realmente se ejecuta es

```
$ ls -FC
```

Esta *expansión de apodo* sucede en todos los usos de **ls** que se hagan sin especificar el nombre de camino completo, de modo que

```
$ /bin/ls
```

se interpreta exactamente tal y como se teclea, pero

```
$ ls -l
```

será interpretado como

```
$ ls -FC -l
```

cuando se ejecute. Ya que esta expansión de apodos es manejada directamente por el shell sin requerir la ejecución de un subshell, es muy rápida. Los apodos te permiten adaptar la sesión y las órdenes a tu gusto sin excesivo recargo para el sistema.

Muchos otros shell han sido desarrollados para propósitos especiales. Uno de los tipos más comunes proporciona una interfaz de usuario de pantalla total u orientada a ventanas, que ofrece menús para ejecución de las órdenes habituales, selección de función con ratón, ayuda en línea y otras funciones de ayuda al

usuario. Shells especiales son utilizados a menudo para facilitar las tareas de administración del sistema o como frontales [front-ends] para paquetes de aplicación. Tales shells son frecuentemente denominados *agentes de usuario*, y las funciones que proporcionan varían ampliamente de una implementación a otra. Con frecuencia son ofertados por los vendedores de software y no forman parte oficial del sistema UNIX.

A continuación se ofrece una visión general y bastante resumida acerca de los distintos tipos de shell:

Nombre del shell	Comando	Descripción
Shell Bourne	<i>Sh</i>	Disponible en todos los sistemas UNIX
Shell C	<i>Csh</i>	Usa una sintaxis similar al estilo de programación C
Shell Korn	<i>Ksh</i>	Superconjunto del shell Bourne
Shell Bash	<i>Bash</i>	Contiene características de los shell Korn y C
Shell Tcsh	<i>Cts.</i>	Funciona como el shell C

Como cada shell es una utilidad del sistema, se puede empezar utilizando un shell con sólo ejecutar el comando que aparece en la tabla anterior.

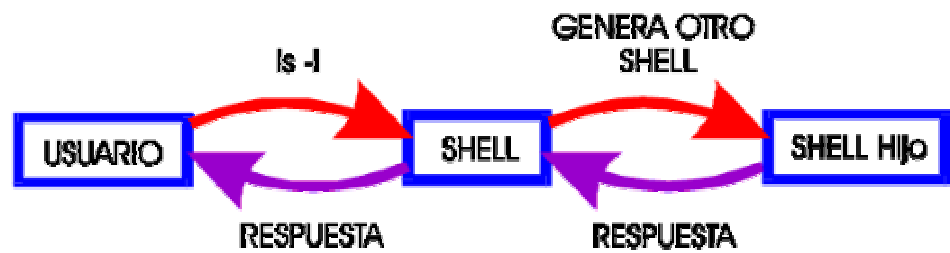
Dependiendo de los shell que usemos, tendremos un indicador de órdenes u otro. Así,

- El shell Bourne y el Korn nos proporcionan el indicador **\$**.
- El shell C nos proporciona **%**.

Para salir de un shell y regresar al anterior, teclearemos **exit**. Si abandonamos el shell inicial, nos desconectaremos del sistema, y si es un subshell, regresaremos al anterior.

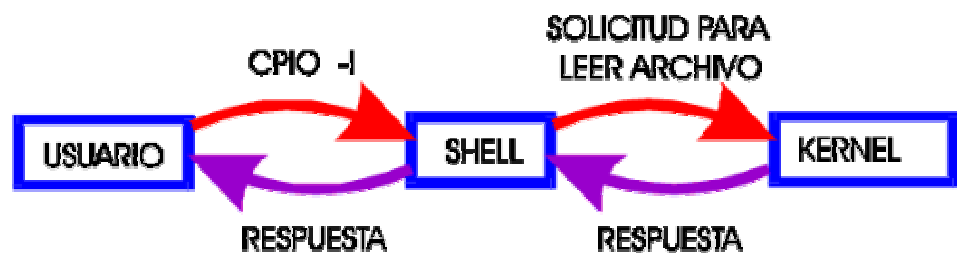
4.4.- Generación de un shell padre o un shell hijo

El shell genera un sub-shell para la ejecución de cada orden. Tras completarse se vuelve al shell “padre”. Si se precede una orden por punto (.), seguido de un espacio, o se agrupan varias órdenes entre llaves en lugar de entre paréntesis, la ejecución se efectúa en el shell actual, sin crear un nuevo shell hijo.



Si el usuario está ejecutando una aplicación, por ejemplo una hoja de cálculo, el usuario está corriendo un shell y el shell está corriendo la aplicación. Un proceso más se agrega entre los dos shells en el caso de una aplicación.

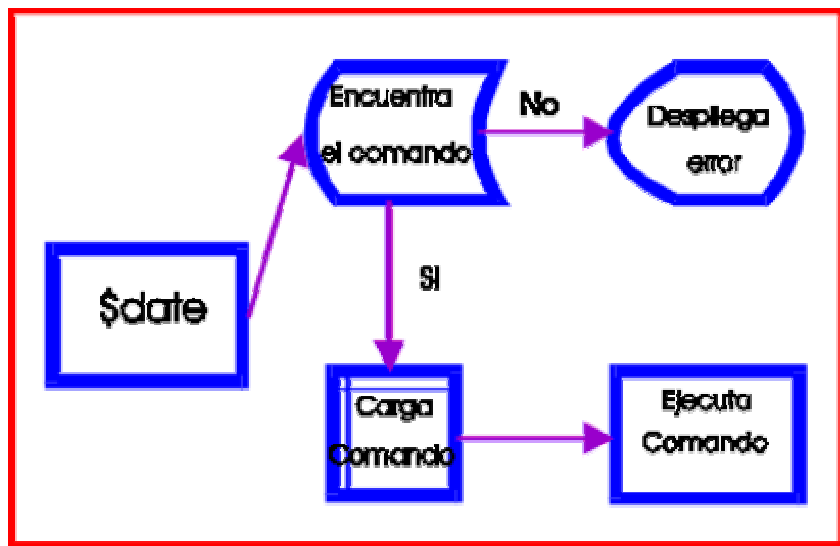
Si la solicitud requiere de la interacción de una componente de hardware, entonces el shell debe involucrar al kernel. A continuación la figura muestra una solicitud para leer de una cinta magnética. El usuario hace la solicitud del shell y el shell en turno hace la solicitud al kernel. El kernel regresa información al shell y el shell en turno lo regresa al usuario.



4.5.- Ejecución de Programas

Antes de adentrarnos en las diferentes formas que tenemos para dar órdenes al shell, haremos una pequeña introducción al modo que tiene el sistema de ejecutar los programas. De cualquier forma, se hará una explicación más extensa más adelante.

Como se mencionó anteriormente, el shell es el responsable de cargar y empezar los comandos que usted ejecuta. El shell usa la línea de comandos para recolectar de usted, el nombre del comando, las opciones y los argumentos que le desea pasar al comando. El shell mediante varios pasos ejecuta el programa según ilustramos en la figura.



Aquí el usuario escribe un comando, el shell localiza el comando buscando en cada uno de los directorios de la variable **PATH**. Si el comando no se encuentra, el shell despliega un mensaje de error , pero si se encuentra entonces el shell determina si el usuario lo puede ejecutar y de ser así lo carga en la memoria y le pide al kernel que lo ejecute. El resultado es la salida del comando como se esperaba.

El shell es un comando también, queriendo esto decir que al ejecutar el comando **sh** usted está obteniendo un subshell. Un subshell hereda el ambiente del padre, pero no así las variables locales del padre.

4.6.- Distintos métodos para generar una línea de órdenes única para el Shell

* Agrupando la línea entre paréntesis y pulsando la tecla **[INTRO]** antes de cerrarlo. Ej,

```
$ (echo "Hola [INTRO]  
>buenos días") [INTRO]
```

Al generar una nueva línea de pantalla sin completar una línea de órdenes aparece el indicador o prompt secundario >. En cada nueva línea de pantalla aparecerá un nuevo indicador > hasta completar la línea de órdenes.

* Agrupando las diferentes líneas de pantalla (generada cada una al pulsar **[INTRO]**) entre llaves. Ej,

```
$ {cal [INTRO]  
>who [INTRO]
```

```
> date ;} [INTRO]
```

```
...
```

(salida de ejecución)

Es preciso dejar algún espacio tras la llave **{** para que el shell no lo interprete como el primer carácter de la orden **cal**. El punto y coma final antes de la llave de cierre **}** es preceptivo para que no se interprete **{** como una opción o argumento de la última orden.

* Evitando la interpretación del carácter de nueva línea precediendo la pulsación de **[INTRO]** por el carácter de “escape” ****. Ej,

```
$ echo hola \ [INTRO]
```

```
< buenos dias [INTRO]
```

4.7.- Inductor de Órdenes o Prompt

Es el símbolo **\$**, el **%** o **#** (para el administrador). Indica la disposición para recibir instrucciones. Si no está presente en pantalla no se pueden introducir nuevas órdenes.

4.8.- Línea de órdenes

Es la unidad de tratamiento e interpretación admitida por el shell. Termina con un carácter de nueva línea **[INTRO]**. Hasta que no termina de completarse, el shell no genera una respuesta.

El shell interpreta los elementos de la línea antes de ejecutar las órdenes que contiene. Es el shell quien interpreta los argumentos, no la orden.

Cuando tecleamos una orden como **cat** al shell, frecuentemente le añadiremos *argumentos* en la línea de orden. Éstos son las opciones modificadoras que comienzan generalmente con un signo **-** (menos) además de los argumentos primordiales tales como nombres de fichero u otros **id** de presentación de usuarios. Las órdenes son de esta forma:

```
$ pr -d nota
```

La orden **pr** se emplea para imprimir ficheros. En realidad la salida va al terminal en lugar de a una impresora, de modo que **pr** actúa de forma parecida a la orden **cat**. Mientras **cat** es mejor para concatenar múltiples ficheros, **pr** es mejor para paginar y sacar los ficheros.

En la sintaxis de las órdenes del sistema UNIX, las opciones siguen al nombre de la orden y preceden a los argumentos primordiales. La orden dada anteriormente dice a **pr** que copie el fichero **nota** en el terminal, haciendo una copia a doble espacio del fichero de entrada. Un nombre de fichero aparece como argumento primordial y la función de la orden está modificada con la opción **(-d)** en este caso, para indicar doble espacio). Ni la opción ni el argumento nombre de fichero son necesarios en la mayoría de las órdenes.

Si lo deseamos, podemos añadir una segunda opción a la orden **pr**:

\$ **pr -n -d** nota

Esto pide a **pr** que genere una salida a doble espacio (**-d**) y numerada por línea (**-n**). Todas las opciones preceden a todos los argumentos nombre de fichero. Una forma equivalente de esta orden es la siguiente:

\$ **pr -nd** nota

Se pueden combinar las opciones con un solo **-** para marcar comienzo de las opciones. La opción **-n** de la orden **pr** puede tener un argumento por sí misma, que signifique el intervalo entre los número que asocia a cada línea. Así se escribe **pr -n2** para numerar las líneas de dos en dos en lugar de uno en uno.

4.9.- Uso de los Metacaracteres

Es importante ser precavido en el uso de metacaracteres o comodines para evitar efectos no deseados. Por ejemplo, la expresión **mm t*** eliminaría del directorio actual todos los archivos cuyo nombre empiece por **t**; pero la expresión **mm t *** eliminaría el archivo **t** y, a continuación, todos los archivos del directorio.

El shell usa comodines para generar grupos o rangos de caracteres:

- i) **?** ➔ sustituye a un único carácter (cualquiera).
- ii) ***** ➔ sustituye a cualquier carácter o grupo de caracteres, salvo separadores.
- iii) **[..]** ➔ sustituye cualquier valor incluido entre los corchetes. Ej,

a[1,2,3], daría como resultado la búsqueda de **a1,a2,a3**

a1[xy]3, daría como resultado la búsqueda de **a1x3, a1y3**

Admite rangos especificando los extremos separados por un guión. Así, **a[1-5]** representa **a1,a2,a3,a4** y **a5**.

[A-z] representa todos los caracteres comprendidos entre **A** y **z**, y ello incluye signos de puntuación, asteriscos, etc. Sin embargo, la expresión **[10-25]** sólo incluye **1,5** y los caracteres comprendidos entre **0**, y **2** (**0, 1 y 2**).

- iv) **!** ➔ este signo, tras el corchete de apertura, significa inversión en la selección. Por ejemplo, **a[!xyz]** agrupa todos los nombres constituidos por el carácter **a** y otros caracteres distintos de **x,y** y **z**.

4.10.- Agrupamiento de Órdenes

Pueden introducirse varias órdenes en una sola línea de órdenes de las siguientes formas:

a) **ord1;ord2... → date;man;cal**

Esto provoca la ejecución sucesiva de todas las órdenes especificadas.

b) **(ord1;ord2...) → (date;man;cal)**

Tiene el mismo efecto que el caso anterior, pero puede considerarse un conjunto para ciertas actuaciones

(ord1;ord2;ord3)>arch

Redirige las salidas de las tres órdenes a un archivo **arch** en lugar de a la salida estándar.

(ord1;ord2;ord3) &

Ejecuta las tres órdenes en segundo plano, liberando el terminal para otras tareas.

c) **{ord1;ord2;ord3} → {date;man;cal}**

Ejecuta las tres órdenes en el shell actual, sin generar otros nuevos. Los cambios en el entorno permanecen.

d) **ord1 && ord2 → banner && cal**

Ejecuta **ord2** sólo si la ejecución de **ord1** ha tenido éxito.

e) **ord1 || ord2 → banner || cal**

Ejecuta **ord2** sólo si la ejecución de **ord1** no ha tenido éxito.

f) **ord1 | ord2**

Ejecuta **ord1** y encauza o envía su salida como entrada de **ord2**.

4.11.- Eliminación de Significados Especiales

Algunos caracteres como *****, **&**, **\$**, etc. tienen una interpretación especial para el shell. Si se quiere evitar esta interpretación y utilizarlos como caracteres ordinarios, se pueden emplear los siguientes métodos:

- **\ Barra inclinada invertida.** Inhibe la interpretación del carácter siguiente (solo uno). Ejemplo, *****
- **'..' Comillas simples.** Impide que el shell interprete todo lo incluido entre las comillas
- **".." Comillas dobles.** Evita la interpretación de los caracteres incluidos salvo **\$**, **|**, **"**, **'** (comillas dobles y simples).

- **`..' Acentos graves.** Provoca la ejecución de la orden incluida.

Las palabras cuyo primer carácter es **\$** son interpretadas por el shell como variables de entorno (se explicará más adelante), y se muestra su valor o contenido en lugar de su nombre o identidad.

A continuación aparece un cuadro donde se muestra una lista de caracteres con significados especiales para el shell:

CARÁCTER	SIGNIFICADO
*	Comodín de cualquier grupo de caracteres
?	Comodín de cualquier carácter simple
[..]	Rangos de selección
;	Separador en el agrupamiento de órdenes
<	Redireccionamiento de entrada
>	Redireccionamiento de salida
>>	Añade la salida al “ archivo ” reseñado
2 >	Envía los mensajes de error al archivo indicado
2 >>	Añade los errores al archivo especificado.
	Encauzamiento de órdenes
&	Ejecución en segundo plano
=	Asignación de valores a variables
\$	Valor de contenido de una variable
`..`	Ejecución previa de la orden incluida
\	Eliminación de interpretación del carácter siguiente
‘..’	Eliminación de interpretación de todo lo incluido
“..”	Eliminación de interpretación salvo \$\ “ “

Alguna órdenes de *Propósito General*

En este capítulo afrontaremos la explicación de algunas órdenes básicas y que, sin tener una importancia relevante dentro del sistema, nos permiten realizar ciertas acciones muy simples

Cuando se ejecuta una orden, el sistema establece un *entorno* de ejecución para la orden y le pasa algunas de las variables de entorno. Las órdenes usan frecuentemente este entorno. Por ejemplo, los editores de pantalla total usan la variable del entorno *TERM*.

No todas las variables del entorno están disponibles para las órdenes que ejecutemos. Cuando se ejecuta una orden, el shell le cede un entorno inicial que puede ser transmitido a los subprogramas ejecutados desde ese programa. Sólo aquellas variables del entorno que sean *exportadas* estarán disponibles a los subprogramas a través de este mecanismo. Se trata de un subconjunto de la lista completa de variables de entorno.

5.1.- La orden *banner*

El sistema proporciona muchas órdenes de propósito general, y aunque algunas de ellas son más divertidas que útiles, cada una tiene su lugar en el sistema. Una de las más sencillas es la orden **banner**, que despliega sus argumentos a gran tamaño y los escribe en la salida estándar. Por ejemplo,

```
$ banner Hola mundo
```

```
$
```

La orden **banner** formará líneas separadas en su salida apoyándose inteligentemente en las fronteras de palabras individuales para mantener la visualización en pantalla relativamente sensata. Se utiliza a menudo en funciones de salida tales como los *spoolers* de impresión para visualizar diferentes tipos de información. Sus parámetros son los siguientes:

banner [-w *tamaño*] texto

-w *tamaño* Donde *tamaño* es el tamaño que deseamos para la presentación

“” Si no especificamos el texto entre las comillas, cada palabra aparecerá en una línea

Por ejemplo, la orden

```
$ banner -w20 "Hola mundo"
```

presentaría en pantalla el texto **Hola mundo** en una sola línea y con una anchura de 20 pixeles.

5.2.- La orden *date*

Las excelentes facilidades de mantenimiento del tiempo del sistema UNIX controlan el tiempo dentro del sistema para todas las resoluciones desde milisegundos hasta años. Todos los ficheros tienen asociada una estampación de tiempo de modificación.

Por omisión, la fecha se visualiza de acuerdo con la hora y zona horaria local, y el sistema puede determinar el comienzo y el final del tiempo de ahorro de luz diaria.

La orden **date** puede formatear su salida de diferentes formas y se utiliza para establecer el tiempo del sistema. Puesto que del tiempo correcto depende tanta actividad del sistema, la fecha debería ser siempre correcta. Aunque parezca una niñería, esto es algo que el administrador debería tener en cuenta. Recordemos que órdenes como **at** y **crontab** usan la fecha del sistema. Estas órdenes serán tratadas posteriormente.

Tenemos varios modificadores para la orden **date**. Así, la orden sin argumentos ni parámetros nos visualiza la fecha actual del sistema:

```
$ date
```

```
Wed Jun 10 18:50:06 CEST 1987
```

Se visualiza la fecha y hora actual tal y como es conocida por el sistema.

La sintaxis de esta orden también puede ser:

	<i>[+ formato]</i>
date	<i>mmddHHMM[aa]</i>
%a	Nombre abreviado del día de la semana
/A	Nombre completo del día de la semana
%b	Nombre abreviado del mes
%B	Nombre completo del mes

%c	Formato estándar para la fecha y la hora
%d	Día del mes
%D	Datos del formato %m/%d/%y
%H	Datos de la hora (24 horas)
%l	Datos de la hora (12 horas)
%j	Día del año
%m	Número del mes
%M	Minutos
%n	Avance de línea
%p	Indicación p.m. o a.m.
%S	Segundos
%t	Caracteres del tabulador
%T	Indicación de la hora en formato %H:%M:%S
%w	Día de la semana (0 corresponde a domingo)
%W	Semana del calendario
%y	Las dos últimas cifras del año
%Z	Nombre de la zona horaria.

Para la forma **date** **[+[formato]]** deberemos usar el signo (+) siempre y , a continuación introducir uno de los elementos expuestos anteriormente. Incluso, podemos colocar un literal, del tipo:

```
$ date +"Mes: %m%nDía: %d%nHora: %T"
```

```
Mes: 06
```

```
Día : 10
```

```
Hora : 19:02:58
```

```
$
```

```
$ date + « Hoy es %d. La hora exacta es %r"
```

Hoy es 10. La hora exacta es 06 : 26 : 32 PM

\$

En el segundo modo (**date mmddHHMM[aa]**) se intenta establecer de nuevo la hora del sistema. Naturalmente, esto sólo puede llevarlo a cabo el usuario “administrador”. Así,

```
# date 06071647
```

```
Sun JUN 7 16:47:00 CEST 2002
```

#

Si fuera necesario, podremos añadir un año de dos dígitos al final del argumento para establecer también el año. Así,

```
# date 0607164799
```

daría la misma fecha, pero del año 1999.

En la práctica es recomendable no cambiar la fecha, a menos que se tengan razones poderosas para ello; las discontinuidades en la cuenta del tiempo pueden provocar problemas con algunos programas.

Las órdenes **date** se usan con frecuencia en guiones shell para producir una visualización o para asignarlas a variables de entorno.

La orden **date** funciona en tiempo local, y convierte correctamente la fecha y la hora a GMT (Greenwich Meridian Time) el formato interno del sistema. La zona horaria y la diferencia en horas entre ésta y GMT está almacenada en el fichero **/etc/TZ**. Estas entradas tienen un formato como **EST5EDT**, donde el nombre de la zona horaria va primero (**EST**), seguido del número de horas de diferencia respecto a **GMT** (5), seguido del nombre de la zona horaria (**EDT**).

5.3.- La orden *cal*

Esta orden cae en la categoría de “órdenes divertidas”. Dada, sin argumentos, presenta un calendario del mes actual sobre su salida estándar.

Si se especifica un año como argumento, **cal** imprimirá un calendario completo para ese año. Cualquier año desde 1 hasta 9999 puede ser visualizado. Además, **cal** puede tomar un segundo argumento antes del año, especificando un mes de ese año. Así, la orden

```
$ cal 9 1755
```

visualizará un calendario para septiembre de 1755, aunque a simple vista no pueda tener nada de interesante.

5.4.- El comando *echo*

El comando `echo` sirve para escribir mensajes en la salida estándar. El mensaje se especifica como parámetro. Para escribir mensajes con caracteres especiales (p.e. espacios) debemos encerrarlo entre comillas.

Sintaxis

echo [*opciones*] mensaje

Un mensaje formado por varias palabras puede especificarse sin comillas, pero cada palabra se debe separar sólo por un espacio, ya que cada palabra se considerará un parámetro distinto, y `echo` lo que hace es escribir cada parámetro separado un espacio del anterior.

Después del mensaje, `echo` imprime un retorno de carro (comienza una nueva línea). Para evitar esto, podemos utilizar el parámetro **-n**.

Veamos los siguientes ejemplos. Observamos cómo en el último caso, el prompt aparece inmediatamente a la derecha del mensaje porque hemos empleado **-n**.

```
$ echo Esto son 3 letras: a b c
Esto son 3 letras: a b c
$ echo "Esto son 3 letras: a b c"
Esto son 3 letras: a b c
$ echo -n "Esto son 3 letras: a b c"
Esto son 3 letras: a b cRemigio:~$
```

5.4.- El comando *clear*

El comando **clear** sirve para limpiar (borrar) la pantalla. No admite parámetros.

Sintaxis

clear

5.5.- El comando *reset*

El comando **reset** sirve para borrar la pantalla y reiniciar el terminal. No admite parámetros. Resulta útil cuando el terminal pierde su configuración habitual, por ejemplo, cuando aborta o falla un programa que haya configurado el terminal de forma especial.

reset

5.6.- El comando *man*

El *Manual del Usuario* es la documentación oficial del sistema UNIX y ha sufrido un gran desarrollo y modificación a lo largo de los años. Todas las órdenes, argumentos para órdenes, bibliotecas de subrutinas, formatos de fichero, utilidades y herramientas están documentadas completamente en el *Manual del Usuario*.

El documento original del *Manual* contenía ocho secciones principales, que aún hoy se conservan:

- 1.- Órdenes y Programas de Aplicación
- 2.- Llamadas al Sistema.
- 3.- Subrutinas.
- 4.- Formatos de Fichero.
- 5.- Misceláneas.
- 6.- Juegos.
- 7.- Ficheros Especiales.
- 8.- Procedimientos de mantenimiento del sistema.

Aunque en un principio este manual fue publicado en papel, y contenía toda la información disponible para UNIX en ese momento, posteriormente se incluyeron dentro de los propios sistemas los contenidos enteros de dicho manual. El texto generalmente ocupa de 2 a 3 Megabytes.

Podemos visualizar estas páginas de manual en línea con la orden **man**, que escribe la página **man** solicitada en su salida estándar. Se invoca con un argumento que especifica la página **man** que se desea visualizar.

```
$ man dic
```

visualizará la página **man** para **dic**. Si aparece más de una entrada bajo ese nombre en más de una sección del manual, se visualizarán todas las páginas **man**, una tras otra. Podemos redirigir la salida a un fichero para su examen posterior, si así lo deseamos. Ya que las páginas **man** están con frecuencia formateadas con **nroff** antes de ser visualizadas, la orden **man** puede tardar un tiempo antes de mostrarse.

man *[sección] [opciones]* Tema

[sección]	Delimita el número de secciones para la salida de la orden
-d	Modificará el camino de búsqueda por omisión al dir actual en lugar de al estándar del sistema
-T	Especifica el terminal por dónde se producirá la salida
Tema	Se refiere a la orden de la que buscamos información.

La visualización de temas se hace mediante el comando `less`, por lo que podemos utilizar las mismas teclas para desplazarnos por el texto.

El manual contiene información no sólo de comandos, sino de ficheros, funciones del lenguaje C, funciones de otros lenguajes, etc. Toda esta información se almacena en distintas secciones. Normalmente no nos importa en qué sección aparece un tema, pero a veces el mismo tema aparece en distintas secciones. Por ejemplo, la información del comando `mount` puede confundirse con la información de la rutina de C `mount()`. En tal situación quizá sea necesario indicar en qué sección del manual se encuentra el tema que nosotros queremos ver. Las secciones se identifican mediante un número. Por ejemplo:

Para ver la página del comando `mount`: `man 8 mount` o simplemente `man mount`.

Para ver la página de la rutina de C `mount()`: `man 2 mount`.

Dentro de la información del manual, a menudo se hace referencia a otras páginas del manual, indicando el nombre del tema y poniendo entre paréntesis el número de la sección.

Si no se especifica la sección, **man** buscará en todas las secciones y mostrará el tema de la primera sección que lo contenga. Si se especifica alguna sección, **man** sólo buscará en esa sección.

Por ejemplo, para producir una salida de **man** sobre un terminal `vt100`, o sus réplicas, la orden a usar sería,

```
$ man -Tvt100 1 man
```

Cuando están presentes, las páginas **man** están localizadas en el directorio `/usr/man` y sus subdirectorios. Generalmente hay varios subdirectorios inmediatos de `/usr/man`, siendo los principales **u_man** (por manual de usuario); **p_man** (por manual del programador); y **a_man** (por manual del administrador).

Cada uno de estos directorios incluye subdirectorios para cada sección incluida en esa parte del manual: **man1** (para las órdenes de la Sección 1); **man2** (para las de la Sección 2); y así sucesivamente hasta **man8**. Las páginas **man** están generalmente almacenadas en su formato fuente **troff** original y deben ser formateadas con las herramientas de preparación de textos **troff -man** o **nroff -man** antes de que tomen la apariencia del manual impreso. Se mantiene en formato fuente de modo que pueden ser formateados para cualquier impresora o terminal según sea necesario.

Además, algunos sistemas mantendrán un conjunto de las páginas **man** en ficheros preformateados, de modo que puedan ser accedidos en línea sin el retardo y uso de CPU exigidos por el formateo de las páginas en demanda. Si es así, los directorios se denominan generalmente **cat1** hasta **cat8** en los directorios **u_man**, **p_man** y **a_man**. El tiempo de respuesta para la orden **man** es mucho más rápido cuando las páginas **man** están preformateadas, pero, naturalmente, las páginas preformateadas están preparadas generalmente por el administrador del sistema para los terminales o impresoras menos potentes del sistema.

5.7.- El comando *help*

Es una utilidad de *ayuda* en línea que puede sustituir con frecuencia al manual en un apuro. Esta facilidad ocupa mucho espacio en disco.

El nombre “help” fue usado por primera vez como una pequeña parte de la herramienta de desarrollo software **sccs**. Las revisiones del sistema UNIX anteriores a SVR3 tienen esta utilización, y la facilidad discutida aquí es un paquete completamente nuevo que comparte muy poco con la versión antigua.

Se entra al subsistema de ayuda con la orden **help**, que visualiza un menú para que indiquemos la próxima acción.

Dentro del subsistema de ayuda podemos tomar acciones adicionales de acuerdo con los inductores. Para terminar y regresar al shell, introduciremos **q** tras el inductor “Enter choice >”. También podemos copiar los contenidos de la pantalla a un fichero o utilizarla como entrada estándar para una orden si introducimos **r** tras el inductor.

En este punto podemos introducir > seguido de un nombre de fichero o | seguido de una línea de cauce para redirigir la pantalla actual a un fichero u orden, respectivamente.

```
Enter choice > r
```

```
Enter >file, |cmd(s), or RETURN to continue > >/usr/remigio/help.menu
```

```
Enter choice >
```

En este ejemplo la salida fue redirigida al fichero **/usr/remigio/help.menu**. La acción es efectuada y el sistema de ayuda vuelve al nivel actual para otra selección de menú. Sólo se salva el tema actual, generalmente una única página de pantalla.

Cada submenú de ayuda permite esta redirección, de modo que es fácil atrapar los datos para un uso posterior.

Este menú de nivel superior proporciona acceso a todas las otras capacidades de ayuda. Introduce **s** aquí para ver información iniciadora [starter], y el sistema responderá con otro menú. La sección iniciadora es un modo excelente para los nuevos usuarios de familiarizarse con el sistema UNIX. Las órdenes y características más sencillas del sistema están disponibles bajo la selección de menú **c**. La selección **d** proporciona una bibliografía de documentos útiles, aunque las fuentes listadas en esta salida pueden ser difíciles de localizar. La selección **l** proporciona información acerca de las capacidades y características locales específicas de esta máquina. El administrador del sistema podrá añadir datos a esta sección usando la orden **helpadm**. La opción **t** lista otras ayudas didácticas disponibles en línea. Podemos regresar al menú de nivel superior en cualquier momento introduciendo **h** desde cualquier menú dentro del sistema de ayuda.

Si seleccionamos **l** desde el menú de nivel superior, el sistema de ayuda nos ayudará a localizar órdenes al introducir palabras claves que describan la acción que deseamos realizar. Seleccionamos **k** en este submenú si deseamos que el sistema busque algunas palabras claves que nosotros introduzcamos. Aunque útil, el sistema de ayuda no puede sustituir al *Manual del Usuario*.

Si seleccionamos **u** en el menú de nivel superior, obtendremos información específica de cómo utilizar una orden, incluyendo ayuda sobre los argumentos disponibles y cómo son usados. Si introducimos el nombre de una orden, el sistema nos indicará que seleccionemos la información en la que estamos interesados. Seleccionamos **d** para una descripción de la orden, **e** para algunos ejemplos, y **o** para una lista de las opciones de orden.

Frecuentemente, el material de estas secciones se extiende a más de una página de pantalla, y podemos seleccionar **n** (por página siguiente) cuando aparezca "MORE" en la parte inferior de la pantalla. Como es habitual, podremos redirigir el manual a un fichero o a una orden para examinarla posteriormente.

La selección **g** desde el menú de nivel superior proporciona un glosario de términos y caracteres especiales. Al introducir un término o carácter, el sistema de ayuda devolverá una breve pero útil definición no disponible en el *Manual del Usuario*.

En lugar de seleccionar una opción desde el menú de nivel superior del sistema de ayuda, podemos introducir una orden directamente en el shell para entrar en uno de los subsistemas de ayuda. Así,

```
$ help ls
```

5.8.- El comando *what is*

El comando *what is* (*what is?*) sirve para visualizar descripciones de palabras clave de temas relacionados con el shell, lenguajes de programación, comandos, etc.

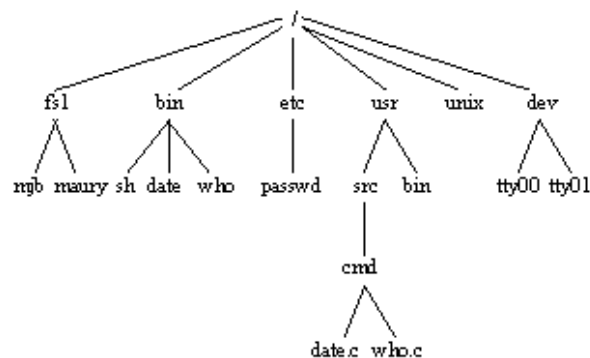
El Sistema de Ficheros

*Otra contribución importante de UNIX es la del sistema de ficheros. La gestión de ficheros en UNIX es extremadamente flexible y potente; tan es así que los conceptos de sistemas de ficheros han sido adoptados en otros sistemas operativos, como en MS-DOS. El sistema UNIX proporciona un esquema de **directorios jerárquicos**. Un directorio es un contenedor para un grupo de ficheros. Las órdenes, los ficheros de datos, otros directorios e incluso los dispositivos hardware pueden ser representados como entradas de un directorio. El sistema UNIX proporciona un modo poderoso pero sencillo de nombrar un fichero o directorio específico dentro del sistema de ficheros, de modo que las órdenes puedan entender fácilmente dónde hallar las cosas que se describan mediante nombres de ficheros en una línea de orden.*

6.1.- Ficheros y Directorios

El sistema operativo UNIX está diseñado para manejar información contenida normalmente en discos. Para que esta manipulación sea realmente efectiva, es necesario que la información esté organizada de alguna forma. La manera estándar de organizar la información es en archivos. Los archivos son localizados dentro del disco porque son apuntados desde un lugar determinado, a este lugar se le denomina directorio.

Sin embargo en UNIX no se utiliza un único directorio para apuntar a todos los archivos del sistema, sino que se crea una estructura jerárquica de directorios conocida como estructura en árbol. Se muestra un ejemplo de esta estructura jerárquica en la figura 3.



Aunque en un principio pueda parecer una estructura excesivamente complicada por la amplia ramificación de los directorios, esta organización permite agrupar los archivos de los diferentes usuarios e incluso de las diferentes aplicaciones en directorios separados, con lo que se evita el que se interfieran entre sí. Por otro lado, un mismo usuario puede organizar su propia información separando los archivos en diferentes directorios de acuerdo a su contenido.

6.2.- Concepto de *archivo*

La estructura fundamental que utiliza el UNIX para almacenar información es el archivo. El UNIX mantiene la pista de cada uno de los archivos asignándole a cada uno, un número de identificación (i-number) que apunta a la tabla de i-nodos. El usuario, no obstante, indicará cada archivo por medio de su nombre asignado por el propio usuario. El UNIX mantiene toda la información propia del archivo, excepto su nombre, en su inodo correspondiente. El nombre se encuentra en la entrada correspondiente del directorio al que pertenece.

Todos los archivos UNIX son tratados como una simple secuencia de bytes (caracteres), comenzando en el primer byte del archivo y terminando con el último. Un byte en concreto dentro del archivo, es identificado por la posición relativa que ocupe en el archivo. La organización de los archivos en registros o bloques de una longitud fija, corresponde a la forma tradicional de organización de los datos en el disco. El UNIX protege específicamente al usuario del concepto de registros o bloques. En su lugar, el UNIX puede permitir que el usuario divida al archivo en registros, de acuerdo a un byte o conjunto de bytes especiales. Los programas son libres para organizar sus archivos, con independencia de la forma en que los datos estén almacenados en el disco.

El nombre de los archivos, anteriormente, sólo podía tener catorce caracteres; actualmente puede tener centenares. Puede contener o empezar con cualquier letra, dígito, puntuación o espacio, pero en la práctica tendría que evitarse poner espacios o signos de puntuación en los nombres de archivos, ya que dificulta su utilización en la línea de comandos. Todos los nombres de un archivo, dentro de un mismo directorio, deben ser únicos.

6.3.- Concepto de *Directorio*

Prácticamente para todos los efectos, un directorio se comporta como un archivo, con la característica de que sus registros son de longitud fija. A pesar de lo comentado anteriormente, el contenido de los directorios no apunta directamente a los bloques de datos de los archivos o subdirectorios que dependen de él, sino a unas tablas (i-nodos) separadas de la estructura. Es desde estas tablas desde las que se apunta a los bloques físicos de los archivos dentro del disco.

Cada registro en el directorio tiene una longitud de 16 bytes de los que:

- Los dos primeros contienen un apuntador (en binario) a la tabla de los i- nodos (i-number).

■ Los catorce restantes contienen el nombre del archivo o subdirectorio (el número bytes por registro depende de la versión del UNIX, ya que hay versiones en que la longitud del nombre de un archivo puede llegar a 255 bytes).

Cualquier directorio contiene un mínimo de dos entradas:

■ Una referencia a sí mismo (.).

■ Una referencia al directorio de que dependo o directorio padre (..), salvo en el directorio raíz que, al ser el comienzo de la estructura, esta referencia es también sobre sí mismo.

i-NODO	Nombre del archivo
20	.
2	..
45	datos
53	fich
.	.
.	.
.	.

6.4.- Sistema de Archivos

El conjunto formado por la estructura de archivos y directorios y las tablas de inodos, se denomina Sistema de Archivos o File System. Cada sistema de archivos está residente en un soporte físico, normalmente disco magnético, pero si el disco es de gran capacidad puede dividirse en varios discos lógicos o "particiones", cada uno de los cuales sería un sistema de archivos independiente. En el caso de que el sistema disponga más de un disco físico, cada uno será forzosamente un file system. Por lo tanto se puede asociar el concepto de sistema de archivos al de disco lógico o partición de disco.

Es importante no perder la idea de que, aparte de otros componentes que se verán más adelante, cada sistema de archivos dispondrá de su propia tabla de i-nodos que apuntarán a los bloques de datos de los archivos y directorios contenidos en él.

El sistema de archivos que contiene el directorio raíz (normalmente el primer disco o la primera partición del primer disco en su caso) es el principal. En principio, cuando se carga el sistema operativo, la única parte accesible de la estructura es la contenida en el file system principal. Para poder acceder a la información contenida en los restantes sistemas de operación es necesario realizar una operación denominada "montaje". Esta operación consiste en enlazar cada sistema de archivos con directorios vacíos de la estructura principal. A partir de ese

momento, los archivos y directorios de los sistemas de archivos secundarios figurarán dentro de la estructura, colgando del directorio de montaje.

Una vez montado todos los sistemas de archivos, la estructura es única, por lo que el disco o la partición en que esté físicamente un determinado archivo, es transparente para el usuario. La función de montaje y desmontaje de los sistemas de archivo es, normalmente, realizada por el administrador del sistema.

El sistema de archivos, o file system, de UNIX está caracterizado por:

- Una estructura jerárquica.
- Tratamiento consistente de los datos.
- La habilidad de crear y borrar archivos.
- Crecimiento dinámico de archivos.
- La protección de los datos.
- El tratamiento de los dispositivos periféricos como archivos.

Cada sistema de archivos consta fundamentalmente de las siguientes partes:

Bloque de carga:

Este bloque, que es el primero de cada sistema (bloque cero), está reservado para un programa de carga. El bloque cero no tiene ningún significado en el sistema de archivos. Toda la información del sistema comienza en el bloque uno del dispositivo. Sólo se utiliza en el sistema de archivos principal.

Súper bloque:

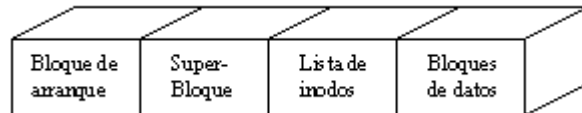
El súper bloque es el bloque uno del dispositivo. Este bloque contiene información sobre el sistema de archivos, tales como su tamaño en bloques, el nombre del sistema de archivos, número de bloques reservados para i-nodos, la lista de i-nodos libres y el comienzo de la cadena de bloques libres. También contiene el nombre del volumen, momento de la última actualización y tiempo del último backup. Siempre reside en un bloque de 512 bytes.

I-nodos:

A continuación del súper bloque están situados los bloques que contienen los i-nodos. El número de bloques de i-nodos varía dependiendo del número de bloques del sistema de archivos. El número de i-nodos está especificado en el súper bloque. Es una tabla que contiene información sobre las características de los archivos. Esencialmente es el bloque de control de los archivos. Hay un i-nodo por cada directorio y archivo del sistema de archivos. El i-nodo contiene una descripción del directorio o archivo, así como el lugar físico que ocupan sus bloques de datos. Los i-nodos sólo apuntan a los archivos o directorios de su mismo sistema de archivos.

Bloques de datos:

El resto del espacio del dispositivo lógico consta de bloques de datos. Bloques de datos que contienen los datos actualmente almacenados en los archivos. Algunos bloques de datos sirven como bloques indirectos, conteniendo números de bloques (direcciones) de grandes archivos.



El sistema de archivos está organizado en una estructura jerárquica. Este sistema jerárquico permite agrupar la información de los usuarios de una forma lógica. Permite manipular eficientemente un grupo de archivos como una sola unidad. No tiene ninguna limitación en su desarrollo, permitiendo un crecimiento dinámico tanto en su número como en su tamaño. El UNIX trata a los archivos de entrada o de salida del mismo modo que cualquier otro archivo. Los usuarios deberán "caminar" a través del árbol hasta encontrar el archivo deseado. Dentro del UNIX el usuario puede acceder a cuatro clases de archivos:

Directorios.

Ordinarios.

Especiales.

Fifo (Pipes).

6.5.- Manejo de los i-nodos

Cuando se intenta acceder a la información contenida en un archivo, el sistema accede al directorio al que pertenece y busca su nombre secuencialmente. Una vez encontrado toma el i-number asociado a ese archivo y con él accede a la entrada correspondiente a la tabla de i-nodos, que contendrá toda la información correspondiente a ese archivo excepto su nombre. Con esta información ya puede acceder físicamente a los bloques de datos dentro del sistema de archivos.

Esta forma de acceder físicamente a los bloques de datos puede parecer más compleja que la utilizada en otros sistemas operativos en los que se accede directamente con la información existente en el directorio, sin necesidad de tablas de i-nodos ni nada por el estilo, sin embargo presenta algunas ventajas. Por ejemplo, si en un directorio se añade una entrada con un nombre de archivo cualquiera, pero con un i-number ya utilizado en otro directorio, se podrá acceder a los mismos bloques de datos desde los dos directorios e incluso con diferentes nombres de archivos. En este caso se dice que existe un enlace entre ellos.

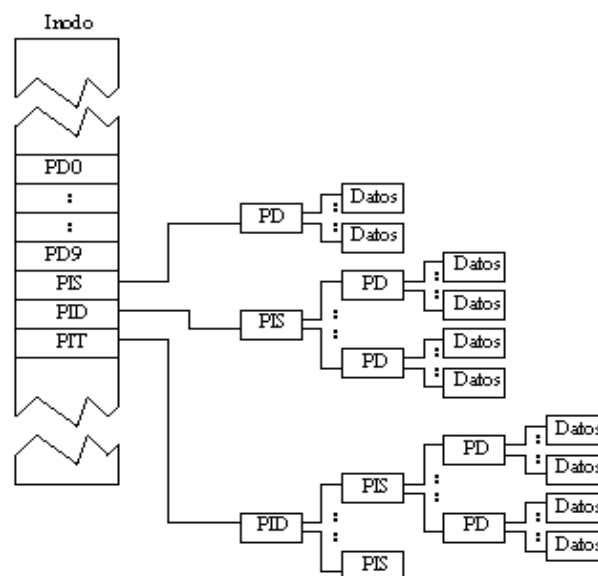
El UNIX mantiene la siguiente información para cada archivo; en la tabla de i-nodos:

- Localización.

- Tamaño.
- Cuántos enlaces tiene el archivo.
- Propietario.
- Permisos de acceso.
- Identificación de si es o no archivo.
- Fecha de creación.
- Fecha de modificación.
- Fecha de último acceso.

Modo
Cuenta de enlace
ID. de usuario
ID. de grupo
Tamaño del archivo
Direcciones de bloque
Fecha de acceso
Fecha de modificación
Fecha de cambio de inodo

Además, cada i-nodo contiene diez punteros directos con la dirección de los diez primeros bloques de datos, un puntero indirecto simple con la dirección de un bloque que contiene punteros directos, que a su vez apuntan a bloques de datos; un puntero indirecto doble que apunta a un bloque con punteros indirectos simples, que apuntan a bloques con punteros directos y éstos a bloques de datos; y un puntero indirecto triple.



6.6.- Archivos Especiales

Los ficheros especiales no sirven para almacenar información, sino para operaciones especiales. Consideraremos ficheros especiales:

☞ Los **enlaces**. Sirven para que podamos acceder al contenido de un fichero o directorio desde distintos puntos del árbol de directorios. Tienen la apariencia de ficheros normales, pero el contenido de estos ficheros es compartido por otros archivos, de forma que si modificamos el contenido del fichero, queda modificado para todos los enlaces, ya que se refieren al mismo fichero.

☞ Los **dispositivos**. Sirven para manejar los distintos periféricos conectados al computador.

En UNIX todos los periféricos se manejan como si fueran ficheros. Por ejemplo, la pantalla se maneja como un fichero llamado `/dev/tty`, de forma que si escribimos información en este fichero, la información, en vez de almacenarse en el disco, se muestra en pantalla. Cada periférico tiene uno o varios ficheros especiales, que llamaremos dispositivos, y todos ellos se colocan en el directorio `/dev`. La tabla siguiente contiene algunos ejemplos de ficheros especiales.

Dispositivo	Significado
<code>/dev/tty</code>	Terminal (consola).
<code>/dev/hda</code>	Primer disco duro
<code>/dev/hda1</code>	Primera partición del primer disco duro
<code>/dev/hdb</code>	Segundo disco duro
<code>/dev/fd0</code>	Primera unidad de disco flexible.
<code>/dev/null</code>	Dispositivo nulo (descarta la información).
<code>/dev/ram</code>	Memoria RAM.

Cuando en UNIX se ejecuta cualquier programa, éste puede utilizar siempre tres dispositivos: la **entrada estándar**, la **salida estándar** y la **salida de errores**.

- La **entrada estándar** es el dispositivo por el que el programa recoge los datos de entrada, la información que debe procesar.
- La **salida estándar** es el dispositivo por el que el programa muestra los resultados, la información procesada.
- La **salida de errores** es el dispositivo por el que el programa muestra los errores durante el procesamiento de los datos.

Por defecto todos estos dispositivos se asocian con la consola, de forma que los datos se introducen por teclado, y los resultados y errores se muestran en pantalla.

6.7.- Principales directorios en UNIX

El camino de directorios que hay que seguir desde la raíz hasta encontrar un fichero se llama ruta (*path*). Para representar la ruta completa de un fichero, se escriben la secuencia de directorios que lo contienen, separados por una barra invertida (/, *slash*), seguido por el nombre del fichero. Por ejemplo, el fichero `libvga.config` que está en el directorio `vga` dentro del directorio `etc`, tiene la ruta `/etc/vga/libvga.config`. El directorio que contiene a otro directorio se denomina **directorio padre** o **directorio superior**. Así, `/etc` es el directorio padre de `/etc/vga`.

Dentro de todos los directorios, excepto del directorio raíz (/), existen dos directorios especiales: "." y ".."

- El directorio `.` es equivalente a la ruta del directorio que lo contiene. Es decir, que es lo mismo `/etc/vga/` que `/etc/vga/./`.
- El directorio `..` es equivalente a la ruta del directorio padre al que lo contiene. Es decir, que es lo mismo `/etc/` que `/etc/vga/../`.

Cuando nos queremos referir a un archivo dentro del árbol de directorios no siempre es necesario especificar la ruta completa del archivo. Para eso existe lo que se denomina **directorio por defecto**, **directorio actual** o **directorio de trabajo**. Cuando el directorio por defecto es *X*, decimos que **estamos** en el directorio *X*, o simplemente que estamos en *X*. En todo momento hay definido un directorio por defecto, de forma que si únicamente especificamos un nombre de archivo, se supone que la ruta hasta ese nombre de archivo es el directorio por defecto.

También podemos especificar directorios a partir del directorio actual. Esto es lo que se denomina una **ruta relativa**, frente a una **ruta absoluta**, que comienza por el directorio raíz (/). Por ejemplo, si el directorio por defecto es `/etc`, entonces son equivalentes `vga/libvga.config` y `/etc/vga/libvga.config`. También podemos utilizar `.` y `..` en rutas relativas. Por ejemplo, si el directorio por defecto es `/etc/ppp`, son equivalentes `../vga/libvga.config` y `/etc/vga/libvga.config`.

Cada usuario de un sistema UNIX tiene un directorio propio, llamado directorio *home*. El directorio *home* es el directorio donde se supone que trabaja el usuario, de manera que puede hacer cualquier cosa en su directorio, y ningún otro usuario puede acceder a él (a menos que explícitamente lo permita). Normalmente el directorio *home* de un usuario tiene el mismo nombre que el usuario y se coloca dentro del directorio `/home`, junto con los de los demás usuarios. Cada vez que un usuario entra al sistema, su directorio por defecto es su *home*. Además, utilizando el comando `cd` sin parámetros, el directorio por defecto volverá a ser el directorio *home*.

En todo sistema UNIX existen un conjunto de directorios que tienen un uso específico. Estos directorios pueden cambiar de unas versiones a otras, pero en general suelen ser los que se muestran a continuación:

- **/tmp** y **/usr/tmp** → se usan para almacenar archivos temporales. Los usuarios también pueden almacenar archivos temporales en ellos, pero tienen que ser muy cuidadosos, ya que el sistema puede borrar periódicamente viejos archivos de los **tmp** sin ningún aviso previo. El tiempo que los archivos permanecen en **/tmp** puede variar de un sistema a otro. A menudo, el sistema borra los archivos de **/tmp** o sus subdirectorios que no se hayan usado durante 7 ó 14 días. Otras veces, si el administrador del sistema tiene poco espacio en el disco, borrará los archivos que se encuentran en este directorio, sin tener en cuenta el tiempo que llevan.
- **/dev** → pertenece al directorio raíz. Contiene el archivo de nodo de dispositivo especial que accede al hardware en el sistema (como, por ejemplo, los disquetes). Estos archivos pueden existir en cualquier otro sitio, pero, de forma convencional, están en este directorio.
- **/bin** → pertenece al directorio raíz. Contiene los programas de utilidad del sistema que son requeridos para las operaciones principales del sistema, como son *ls.bin*. Es la abreviación de “archivos binarios ejecutables”.
- **/usr** → pertenece al directorio raíz, y contiene los archivos del usuario y utilidades del sistema, necesarios para soportar los programas del usuario.
 - **/usr/bin:** contiene las utilidades del sistema necesarias para los usuarios y programas de éstos.
 - **/usr/lib:** contiene librerías de tablas y subrutinas útiles para el sistema.
 - **/usr/spool:** guarda los archivos “spool”, que son copias de archivos temporales colocados en la cola para la impresión, hasta que sean impresos.
- **lost + found** → pertenece al directorio raíz. También existe en otros directorios, en los que el sistema almacena datos de archivos si el nombre del archivo se ha perdido debido a una caída del sistema (recomendamos mirar la sección acerca de la conexión y desconexión del sistema).
- **/home** → Aparece en los nuevos sistemas UNIX (como el *SCO*). Es específico para los directorios personales del usuario, donde pueden guardar sus archivos individuales.
- **/sbin** y **/usr/sbin** → contiene archivos ejecutables específicos para la administración y operación del sistema.
- **/stand** → es especial en algunos sistemas SVR4 para guardar el ‘Kernel’ de UNIX y los archivos de arranque.

- **/var** ➔ contiene archivos con información de variables del sistema; es decir, archivos temporales y registros cuyos contenidos varían con el tiempo. Sustituye al viejo directorio **/var/spool**.

6.8.- Órdenes para el manejo de ficheros y directorios

ls

`ls` significa *list subdirectory*. Sirve para mostrar los nombres de los ficheros y directorios que hay en un directorio. La sintaxis es:

Sintaxis

ls [*modificadores*] [*ficheros*]

Los modificadores permiten especificar diversos formatos y opciones. Los ficheros son la lista de máscaras de los ficheros que hay que listar. Si se omite la máscara, se listarán todos los ficheros. Así, el comando sin parámetros muestra una lista en varias columnas de los ficheros del directorio actual, ordenados alfabéticamente por columnas (de arriba abajo y luego de izquierda a derecha; mayúsculas y minúsculas se ordenan por separado, primero las mayúsculas):

```
Remigio :/boot $ ls
System.map  any_d.b      chain.b      os2_d.b
any_b.b     boot.b       config
```

Podríamos haber indicado una máscara, para listar sólo algunos de los ficheros:

```
Remigio :/boot $ ls *.b
any_b.b  any_d.b  boot.b  chain.b  os2_d.b
```

O también una lista de máscaras:

```
Remigio :/boot $ ls a*.b b*
any_b.b  any_d.b  boot.b
```

En estos listados no se muestran todos los archivos. Los archivos que comienzan por un punto (.) se consideran archivos "especiales", y no se listan normalmente, a menos que especifiquemos lo contrario. Esto lo hacemos con el modificador `-a`:

```
Remigio: /boot $ ls -a
./          System.map  any_d.b      chain.b      os2_d.b
../         any_b.b     boot.b       config
```

Observamos que los nombres de directorios se distinguen de los nombres de ficheros por que aparecen seguidos de una barra inclinada (/). Dependiendo de la versión de UNIX y el tipo de terminal que utilicemos es posible que cada fichero o directorio listado aparezca de un color diferente. Este color indica alguna propiedad del fichero (tipo, permisos, etc.). Con `ls`, además del nombre de los ficheros, podemos obtener información sobre los mismos, con el modificador `-l`.

Los datos proporcionados para cada archivo mediante este formato son los siguientes:

- 1.- Tipo de archivo.
- 2.- Derechos de acceso.
- 3.- Cifras de referencia.
- 4.- Propietario del archivo.
- 5.- Grupo al que pertenece el archivo.
- 6.- Tamaño del archivo en caracteres (bytes).
- 7.- Última fecha de modificación.
- 8.- Nombre del archivo.

```
Remigio:/boot$ ls -l
total 157
-rw-r--r--  1 root    root      143835 Jun 24  1997 System.map
-rw-r--r--  1 root    root         204 May 25  1996 any_b.b
-rw-r--r--  1 root    root         204 May 25  1996 any_d.b
-rw-r--r--  1 root    root        4416 May 25  1996 boot.b
-rw-r--r--  1 root    root         88 May 25  1996 chain.b
-rw-r--r--  1 root    root       5147 Jun 24  1997 config
-rw-r--r--  1 root    root        192 May 25  1996 os2_d.b
```

El código de colores no puede utilizar siempre, ya que es muy habitual utilizar terminales monocromo para conectarnos a las máquinas UNIX. Por eso, en vez del código de colores, podemos utilizar un código de caracteres para determinar el tipo de los ficheros listados. Esto lo hacemos con el modificador `-F` (a veces se utiliza esta opción por defecto, aunque no la especifiquemos):

```
Remigio:/lib$ ls -F
cpp@          libcurses.so.1@      libm.so.5@
ld-UNIX.so@   libcurses.so.1.0.0*  libm.so.5.0.9*
ld-UNIX.so.1@ libdl.so@            libncurses.so@
ld-UNIX.so.1.9.5* libdl.so.1@          libncurses.so.1.9.9e@
ld.so*        libdl.so.1.9.5*      libncurses.so.3.0@
libc.so.4@     libe2p.so.2@         libncurses.so.3.0.0*
libc.so.4.7.6* libe2p.so.2.3*       libss.so.2@
libc.so.5@     libext2fs.so.2@      libss.so.2.0*
libc.so.5.4.33* libext2fs.so.2.3*    libtermcap.so.2@
libcom_err.so.2@ libgdbm.so.1@        libtermcap.so.2.0.8*
libcom_err.so.2.0* libgdbm.so.1.7.3*    libuuid.so.1@
```

libcurses.so.0@	libm.so.4@	libuuid.so.1.1*
libcurses.so.0.1.2*	libm.so.4.6.27*	modules/

El código se añade al nombre del fichero: * indica un fichero ejecutable, / indica un directorio, @ indica un enlace simbólico, etc. Cuando se trata de un archivo normal, no se añade nada.

La tabla siguiente resume cada uno de los modificadores y su función principal:

Modificador	Resultado
-l	Lista los ficheros en una única columna (un fichero por línea).
-b	Los caracteres que no pueden visualizarse se presentarán mediante una barra diagonal y una cifra octal de tres posiciones.
-l	Formato largo: muestra información de los ficheros (premisos, propietario, fecha, etc.)
-m	Todos los nombres de archivo se mostrarán en forma de lista separada por comas.
-n	En lugar del nombre del propietario y el nombre del grupo se indicará el número de usuario del propietario y el número de grupo.
-q	Para los caracteres no imprimibles del nombre de archivo se representará un signo de interrogación (?).
-R	Recurso: para cada directorio, muestra también su contenido de forma recursiva.
-a	Todos los ficheros: incluye los ficheros que comienzan por un punto (.).
-A	Casi todos los ficheros: incluye los ficheros que comienzan por un punto (.), excepto los directorios "." y "..".
-B	Sin <i>backups</i> : no muestra los ficheros de <i>backup</i> ¹ (copia de seguridad), que lo los ficheros que terminan en "~".
-d	Lista el nombre de un directorio en lugar de su contenido.
-t	Ordena los ficheros listados por la fecha y hora de modificación.
-S	Ordena los ficheros listados por su tamaño.
-X	Ordena los ficheros listados por su extensión.
-r	Ordena los ficheros de forma inversa.
-U	No ordena los ficheros de ninguna forma; los lista en el orden en que aparecen en el disco.
-s	Muestra el tamaño de un archivo a la izquierda de su nombre, medido en bloques.
-F	Muestra un carácter al final del nombre, que indica el tipo de fichero (como el código de colores).
-i	Muestra el inodo (la zona de disco) en que se ubica la información del fichero.

¹ Muchas aplicaciones en UNIX que manipulan documentos de datos, cuando salvan un documento, guardan la copia anterior del archivo, con el mismo nombre, pero acabado en "~".

Existen varios comandos derivados de `ls` que listan los ficheros con algunas opciones predeterminadas: `dir`, `vdir`, etc.

pwd

pwd (*print working directory*) muestra la ruta completa del directorio actual. No admite modificadores ni parámetros.

```
Remigio:~$ pwd
/home/Remigio
```

cd

cd (*change directory*) sirve para cambiar el directorio por defecto, permitiendo navegar por el árbol de directorios. Admite como parámetro el nombre del directorio al que queremos cambiar.

Cuando cambiamos a un directorio, podemos hacerlo de forma *relativa* o *absoluta*. Para cambiar a un directorio de forma absoluta debemos conocer la ruta exacta y asegurarnos de que el nombre del directorio comienza por `/`. El directorio actual también se conoce como directorio punto (`.`).

Si usamos la forma **cd ..** subiremos hasta el directorio padre respecto al directorio donde nos encontremos.. Podremos movernos hacia abajo, hacia los subdirectorios, en tanto existan directorios. Al final de la jerarquía de directorios, no habrá más subdirectorios, únicamente ficheros. Eventualmente, si subimos llegará un momento en el que no hayan más directorios superiores. Si usamos **cd** sin especificar directorios, nos lleva al directorio *home*, convirtiéndose éste en el directorio actual. Para conocer el directorio donde nos encontramos usaremos la orden anterior (**pwd**).

Para cambiar a un directorio de forma *relativa* usaremos los dos puntos (`..`) para subir un nivel en el árbol de directorios hasta alcanzar nuestro destino. Así, una posible ruta relativa sería:

```
cd ../usuario3/documentos
```

Se puede especificar una ruta de acceso relativa o completa para cambiar a un subdirectorio del directorio actual. Sin embargo, resulta más fácil hacerlo mediante una ruta relativa. Por ejemplo, supongamos que el directorio */home* tiene los subdirectorios **user1** y **user2**. Y dentro de **user1** tenemos **documentos** y **ejercicios**. Si nos situamos en **home**, para acceder a **user1** haríamos,

```
cd user1
```

y si desde **home** quisiéramos alcanzar **documentos**, haríamos,

```
cd user1/documentos
```

rm

El comando `rm` (*remove*) sirve para borrar ficheros. Admite como parámetro una lista de rutas o máscaras para ser borrados.

Sintaxis

`rm` [*opciones*] ficheros

Cuando no tenemos permiso de escritura sobre los archivos que vamos a borrar, `rm` nos pedirá confirmación. Si queremos omitir la confirmación podemos utilizar el modificador `-f`. por el contrario, si queremos confirmación para cada fichero (aunque tengamos permiso de escritura) utilizamos `-i`.

Algunos ejemplos de uso de `rm`:

```
Remigio:~$ rm .c *.h
Remigio:~$ rm -f /tmp/*
```

Podemos borrar directorios completos con la opción de recursividad `-R`. El siguiente comando elimina todo el contenido de un directorio (y los subdirectorios) sin confirmación: `rm -fR *`. Debemos tener mucha precaución al utilizar este comando, puesto que su acción no puede recuperarse.

NOTA IMPORTANTE:

No debemos confundir `rm` con el comando de renombrar (`mv`). Muchos usuarios de MS-DOS escriben instintivamente `rm` con la intención de renombrar, ya que se relaciona con el comando de MS-DOS `rename`. **Lo que borremos accidentalmente no puede ser recuperado.**

La tabla siguiente recoge algunos de los parámetros más importantes de la orden `rm`:

Modificador	Resultado
<code>-i</code>	Interactivo: pide confirmación antes de borrar cada fichero.
<code>-f</code>	No pide confirmación ni produce error si un fichero no existe.
<code>-r</code>	Recursivo: borrar los ficheros dentro de los subdirectorios y elimina también los subdirectorio.
<code>-R</code>	Igual que <code>-r</code> .
<code>-v</code>	Muestra el nombre de cada fichero antes de borrarlo.

rmdir

El comando `rmdir` (*remove directory*) sirve para borrar subdirectorios vacíos. Podemos borrar varios directorios en la misma orden, especificando una lista. Si un directorio no está vacío, no puede ser borrado. un directorio está vacío cuando únicamente contiene las entradas `"."` y `".."`, que no se pueden borrar.

Sintaxis

`rmdir` [*opciones*] directorios

Podemos utilizar el parámetro `-p` para hacer borrados recursivos, siempre que la ruta indicada no contenga otros ficheros. Por ejemplo, si hacemos `rmdir -p /tmp/1/2/3`, se intentará eliminar `/tmp/1/2/3`; si no está vacío o no se puede borrar, terminará; si no, luego tratará de borrar `/tmp/1/2`, luego `/tmp/1`, y finalmente `/tmp`.

La tabla siguiente muestra los modificadores que podemos usar con `rmdir`:

Modificador	Resultado
<code>-p</code>	Borra recursivamente: después de borrar un directorio, intentar borrar el directorio padre.

mkdir

El comando `mkdir` (*make directory*) sirve para crear nuevos subdirectorios, dando lugar a la estructura del árbol de directorios. Podemos crear varios directorios en la misma orden, especificando una lista; los directorios se crearán en el mismo orden de la lista.

Sintaxis

`mkdir` [*opciones*] directorio

Los nuevos directorios se pueden especificar mediante una ruta absoluta o relativa al directorio actual. Solamente puede crearse un nivel de directorio de una sola vez. Por ejemplo, si no existe el directorio `/tmp/1`, no podemos crear `/tmp/1/2`. Antes debemos crear `/tmp/1`, o bien utilizar la opción `-p`.

```
Remigio:~$ mkdir /tmp/1/2
mkdir: cannot make directory `/tmp/1/2': No such file or directory

Remigio:~$ mkdir /tmp/1 /tmp/1/2
Remigio:~$ mkdir -p /tmp/1b/2b/3b
```

La tabla siguiente resume los principales modificadores del comando `mkdir`.

Modificador	Resultado
-p	Crea recursivamente: antes de crear un directorio, crear su directorio padre (si no existe).
-m	Permite especificar los permisos del nuevo directorio.

mv

El comando `mv` (*move*) sirve para cambiar el nombre de ficheros o directorios, o para moverlos dentro del árbol de directorios. Debemos especificar uno o varios archivos de origen y una ruta de destino. Si sólo hay un origen, el destino puede ser un nombre de fichero –con su ruta correspondiente– y `mv` renombrará y/o moverá el archivo; en cambio, si hay varias rutas como origen, o el origen contiene alguna máscara, el destino sólo puede ser un directorio, y `mv` sólo actúa moviendo los archivos a la ruta indicada como destino.

Sintaxis

`mv` [*opciones*] origen destino

Las opciones que admite este comando se resumen en la tabla siguiente. Son semejantes a las que se utilizan para el comando `cp`.

Modificador	Resultado
-i	Interactivo: pregunta antes de sobrescribir un fichero existente en el directorio de destino.
-b	Crea una copia de seguridad de los archivos que sobrescribe.
-u	Actualizar: no mueve ficheros a menos que el origen sea más moderno que el destino.
-v	Muestra el nombre de cada fichero antes de moverlo.

A continuación vemos los siguientes ejemplos de uso de este comando:

- * Renombrar el fichero `copia1` a `copia2`.
- * Mover y renombrar el fichero `copia1` a `copia2` en el directorio padre.
- * Mover varios ficheros al directorio temporal.
- * Mover varios ficheros del directorio temporal al directorio actual.

```
Remigio:~$ mv copia1 copia2
Remigio:~$ mv copia1 ../copia2
Remigio:~$ mv *.tiff /tmp
Remigio:~$ mv /tmp/*.tiff .
```

cp

El comando `cp` (*copy*) sirve para copiar ficheros. Admite como parámetros un fuente y un destino, de forma que el archivo especificado como fuente se copia en la ruta de destino.

Sintaxis

cp [*opciones*] origen destino

Si la ruta de destino no incluye un nombre de archivo (sólo son nombres de directorios), el nombre del fichero de destino será el mismo que en el origen. Por otro lado, si la ruta de origen es una máscara o una lista de rutas o máscaras, el destino sólo puede ser el nombre de un directorio. Todos los archivos de origen se copiarán al directorio de destino.

Ejemplos:

```
Remigio:~$ cp fichero1 fichero2
Remigio:~$ cp fichero1 /tmp/fichero2
Remigio:~$ cp *.c /tmp
Remigio:~$ cp fichero1 fichero2 /tmp
```

Los archivos recién copiados tienen la fecha actual y como propietario el usuario que realiza la copia, a menos que utilicemos el modificador `-p`. Otras posibles opciones (modificadores) del comando `cp` se resumen en la tabla siguiente:

Modificador	Resultado
-i	Interactivo: pregunta antes de sobrescribir un fichero existente.
-l	En vez de copiar, crea enlaces.
-s	En vez de copiar, crea <i>enlaces simbólicos</i> .
-b	Crea una copia de seguridad de los archivos que sobrescribe.
-P	Copia el archivo de origen en el directorio de destino incluyendo la ruta completa de origen.
-p	Mantiene la fecha, permisos, propietario y grupo de los archivos origen.
-R	Recursivo: copia también los ficheros que estén dentro de directorios.
-u	Actualizar: no sobrescribe ficheros a menos que el origen sea más moderno que el destino.
-v	Muestra el nombre de cada fichero antes de copiarlo.

cat

El comando `cat` (*concatenate*) sirve para encadenar varios ficheros. El resultado se muestra por pantalla (por la salida estándar). Admite como parámetro la lista de ficheros que hay que encadenar. Estos se enviarán a la salida estándar en el mismo orden en que aparecen en la lista. Si sólo se especifica un fichero, el resultado será que se mostrará por pantalla ese fichero (este es uno de los usos más habituales de `cat`). Si no se especifica ningún fichero, `cat` tomará el contenido de la entrada estándar (teclado).

Sintaxis

cat [*opciones*] ficheros

`cat` se utiliza a menudo para componer pequeños ficheros de texto. El comando "`cat >nuevo_texto`" crea el fichero `nuevo_texto` con el contenido de lo que a continuación escribamos. Lo que hace es tomar la entrada estándar (teclado en este caso) y lo pasa a la salida estándar, que en este caso está redireccionada hacia el fichero `nuevo_texto`. Cuando utilizamos el teclado para componer la entrada a un comando, para indicar el final del texto que escribimos tenemos que pulsar `^D` (la tecla *Ctrl* más la tecla *D*). Si no estamos al principio de la línea (hemos escrito algo antes de pulsar `^D`), tendremos que pulsar `^D` dos veces. Por ejemplo (`^D` no aparece en pantalla):

```
Remigio:~$ cat >nuevo_texto
Este es el texto que va a contener
escribimos nosotros)
el fichero.
```

(estas dos líneas las escribimos nosotros)

```
^D                                     (pulsemos ^D)
Remigio:~$ cat nuevo_texto
Este es el texto que va a contener
el fichero.
```

El comando `cat` admite los modificadores:

Modificador	Resultado
-n	Hace que se numeren las línea mostradas (empezando en 1).
-b	Hace que se numeren las línea mostradas, pero no se numeran las líneas en blanco (vacías).
-s	Elimina las múltiples líneas en blanco contiguas por una única línea vacía.
-A	Muestra los caracteres no imprimibles con notación "^", y a los caracteres cuyo código ASCII sea mayor que 127 les antepone "M-"

more

El comando `more` sirve para mostrar un fichero haciendo una pausa en cada página (pantalla). Si especificamos varios ficheros, se mostrarán en el mismo orden, haciendo pausa e imprimiendo una línea de título antes de comenzar cada uno.

Sintaxis

more [opciones] [ficheros]

Para avanzar por el texto paginado, utilizamos las siguientes teclas:

- *Barra espaciadora*: avanzar una página.
- *Enter*: Avanzar una línea.
- *q*: terminar.

Cada vez que avanzamos en el fichero mostrado, las nuevas líneas aparecen en la parte inferior de la pantalla, mientras las anteriores se desplazan hacia arriba, a menos que empleemos la opción `-p`, que hace que las páginas se reemplacen unas a otras sin que se produzca desplazamiento (esto es más rápido). Otras opciones que permite este comando se resumen en la tabla que se muestra a continuación.

Modificador	Resultado
-###	Indica el número ### de líneas por página.
+###	Indica el número ### de línea por el que hay que comenzar.
+/	Seguido de una cadena, hace que se comience a mostrar el fichero por la primera aparición de la cadena.
-P	Hace que las páginas se muestren sin que se produzca desplazamiento (<i>scroll</i>)

less

El comando `less` sirve para visualizar el contenido de un fichero poco a poco. Es semejante a `more`, pero con más funcionalidad. La principal ventaja sobre `more` es que se puede navegar por el texto tanto hacia delante como hacia atrás.

Sintaxis

less [*opciones*] [*ficheros*]

Para avanzar y retroceder por el texto paginado podemos utilizar, además de las teclas de `more`, los cursores. Además existen multitud de teclas y opciones para todo tipo opciones, que podemos consultar con `man less`. Destaca la posibilidad de numerar las líneas, lo que hacemos con `-N`.

El comando `man` utiliza `less` para mostrar su información.

Modificador	Resultado
-N	Muestra el número de línea junto a la línea.

tail

El comando `tail` sirve para mostrar por la salida estándar las últimas líneas de un fichero. Por defecto muestra las 10 últimas líneas, pero podemos seleccionar otro número.

Sintaxis

tail [*opciones*] fichero

La opción más habitual es `-n`, que permite especificar a continuación el número de líneas que hay que mostrar. Por ejemplo, para ver las 3 últimas líneas de un fichero:

```
Remigio:~$ tail -n 3 mi_fichero
```

Si especificamos varios ficheros a la vez, se mostrarán en el mismo orden, con un título antes de cada uno, a menos que empleemos la opción `-q`. Ejemplo:

```
Remigio:~$ tail -n 2 animales plantas
==> animales <==
elefante
rana

==> plantas <==
rosa
alcornoque
Remigio:~$ tail -q -n 2 animales plantas
elefante
rana
rosa
alcornoque
```

La tabla siguiente recoge los principales modificadores que se pueden utilizar con `tail`.

Modificador	Resultado
<code>-c</code>	Permite especificar a continuación el número de caracteres del final que hay que mostrar (en vez de líneas).
<code>-f</code>	Con este parámetro se espera tras la visualización de la cantidad deseada de datos los caracteres y líneas que faltan
<code>-b</code>	La orden tail mostrará la información basándose en una cuenta de bloques.
<code>-q</code>	No muestra títulos antes de cada fichero cuando hay varios ficheros.
<code>-n</code>	Permite especificar a continuación el número de líneas que hay que mostrar (por defecto 10).

head

El comando `head` sirve para mostrar por la salida estándar las primeras líneas de un fichero. Por defecto muestra las 10 primeras líneas, pero podemos seleccionar otro número.

Sintaxis

```
head [opciones] fichero
```

La opción más habitual es `-n`, que permite especificar a continuación el número de líneas que hay que mostrar. Por ejemplo, para ver las 3 primeras líneas de un fichero:

```
Remigio:~$ head -n 3 mi_fichero
```

Si especificamos varios ficheros a la vez, se mostrarán en el mismo orden, con un título antes de cada uno, a menos que empleemos la opción `-q`. Ejemplo:

```
Remigio:~$ head -n 2 animales plantas
==> animales <==
oso
pez

==> plantas <==
pino
encina
Remigio:~$ head -qn 2 animales plantas
oso
pez
pino
encina
```

La **¡Error! No se encuentra el origen de la referencia.** recoge los principales modificadores que se pueden aplicar a `head`.

Modificador	Resultado
<code>-c</code>	Permite especificar a continuación el número de caracteres del principio que hay que mostrar (en vez de líneas).
<code>-q</code>	No muestra títulos antes de cada fichero cuando hay varios ficheros.
<code>-n</code>	Permite especificar a continuación el número de líneas que hay que mostrar (por defecto 10).

WC

El comando `wc` (*word count*) sirve para contar las líneas, palabras y caracteres que contiene un fichero. Por defecto cuanta estas tres cosas. Para cada fichero procesado por el comando, escribe (en la salida estándar) 4 columnas que indican el número de líneas, palabras y caracteres (en este orden) y el nombre del fichero procesado. Además, si se procesan varios ficheros, al final se muestra la cuenta total.

Sintaxis

```
wc [opciones] [fichero]
```

Para `wc` una palabra es el conjunto de caracteres que hay entre dos blancos (espacio, tabulación, retorno de carro, etc). Con las opciones `-c`, `-w` y `-l` podemos hacer que en vez de las tres cuentas sólo se muestre una o dos, tal y como se indica en la tabla siguiente. En cualquier caso el nombre de cada fichero procesado aparecerá a la derecha de las cuentas, y los números aparecen en el orden líneas, palabras, caracteres (independientemente del orden de los modificadores utilizados). El siguiente ejemplo muestra el número de líneas y palabra de los ficheros `*.c`.

```
Remigio:~$ wc -wl *.c
   3      11 f1.c
 533   1118 data.c
   24    121 x_2.c
    0      0 empty.c
 560   1250 total
```

Si no se indica ningún fichero, contará el contenido de la entrada estándar.

Modificador	Resultado
<code>-c</code>	Sólo muestra el número de caracteres (número de bytes).
<code>-w</code>	Sólo muestra el número de palabras.
<code>-l</code>	Sólo muestra el número de líneas.

sort

El comando `sort` sirve para ordenar alfabéticamente las líneas de un fichero, para mezclar las líneas de varios ficheros y para comprobar si un fichero está ordenado. La selección entre estas tres operaciones se hace mediante los modificadores `-c` y `-m`.

Sintaxis

sort [*opciones*]

- Si no se utiliza `-c` ni `-m`, sirve para ordenar: toma los ficheros especificados como parámetro, los fusiona, ordena el resultado y lo muestra por la salida estándar.
- Con `-c`, `sort` se utiliza para comprobar si un fichero está o no ordenado. Si lo está, no hace nada; en caso contrario termina con error.
- Con `-m`, `sort` se utiliza para mezclar los ficheros especificados. No es más que otra forma de ordenar, pero requiere que los ficheros especificados estén individualmente ordenados. Como ventaja, este método es más rápido que la ordenación normal.

El criterio de ordenación por defecto es el orden alfabético basado en el orden ASCII, aunque podemos utilizar otros criterios. El modificador `-f` hace que no se distinga entre mayúsculas y minúsculas, `-n` indica que la ordenación es numérica (y no alfabética), `-b` ignora los espacios antes de la primera letra, etc. La tabla siguiente contiene los principales modificadores.

Habitualmente los ficheros que ordenemos tendrán campos, es decir, estarán formado una tabla organizada en filas (líneas) y columnas (separadas mediante algún carácter, por ejemplo, un tabulador). Entonces nos puede interesar ordenar esa tabla (fichero) por algún campo que no tiene necesariamente que ser el primero de cada línea. Para ello debemos indicarle a `sort` dos cosas:

- El carácter que separa los campos. Para esto utilizamos la opción `-t` seguida del carácter de separación (hay que dejar un espacio después de `-t` y antes del carácter).
- El número de campo por el que ordenamos (la clave de ordenación). Esto lo indicamos con la opción `+++` y `---`, donde `+++` representa el número de campo (comenzando en 0). `+++` indica el primer campo de ordenación y `---` el posterior al último (si se omite éste, se supone hasta el final de la línea). Es decir que la clave son los campos incluidos entre `+++` y `---`, sin incluir éste. El número de campo se especifica como uno o dos números separados por un punto (.); lo que queda a la izquierda es el número de campo; lo que queda a la derecha (la parte opcional) es el número de carácter dentro del campo (si se omite el número de carácter, se debe omitir el punto). Detrás del número (sin separación) se puede utilizar un carácter para especificar opciones de ordenación (numérica, ignorar mayúsculas/minúsculas, etc),

La opción `-o` seguida (con una separación) de un nombre de fichero, hace que el resultado se envíe a ese fichero en lugar de a la salida estándar. Esto permite que se pueda utilizar ese mismo fichero como fichero de entrada, haciéndose una copia temporal para utilizarla como entrada.

Por último, la opción `-r` ordena de forma inversa.

Modificador	Resultado
-b	Ignora los blancos (espacio, tabuladores, etc.) al principio de las líneas.
-c	Verifica si los datos de entrada están o no clasificados. Sólo tendrá lugar una salida si se trata de datos clasificados.
-d	Ignora los caracteres que no sean letras o números.
-f	Ignora la diferencia entre mayúsculas y minúsculas.
-M	Supone que la clave de ordenación representa los meses del año en formato "JAN", "FEB",... "DEC". Además ignora la diferencia entre mayúsculas y minúsculas.
-n	Supone que la clave de ordenación representa un número (entero o decimal).
-r	Invierte el orden.
-o	Permite especificar a continuación un fichero de salida en lugar de la salida estándar.
-t	Permite especificar a continuación el carácter de separación entre campos.
+###	### es el primer campo de ordenación (numerado desde 0).
-###	### es el campo posterior al último campo de ordenación (numerado desde 0).
-u	Si se encuentran varias líneas iguales, éstas sólo se reproducirán una vez.

Ejemplos:

- Listar los ficheros del directorio y ordenarlos según su propietario (con `ls -l` obtenemos un listado en el que la tercera columna es el propietario):

```
Remigio:~$ ls -l|sort +2 -3
total 11
-rw-r--r-- 1 Remigio users          99 Oct 19 14:35 nuevo_texto
-rw-r--r-- 1 Remigio users      7869 Jul 10  1997 termcap
-rw-r-x--x 2 Remigio users        41 Oct 19 14:47 fichero1*
-rw-r-x--x 2 Remigio users        41 Oct 19 14:47 fichero2*
-rwsrwsrwx 1 root      root         0 Oct 19 11:41 p*
```

- Listar los ficheros del directorio y ordenarlos según el mes de creación y después según su tamaño:

```
Remigio:~$ ls -l | sort +5M -6 +4n -5
total 11
-rw-r--r-- 1 Remigio users      7869 Jul 10  1997 termcap
-rwsrwsrwx 1 root      root         0 Oct 19 11:41 p*
-rw-r-x--x 2 Remigio users        41 Oct 19 14:47 fichero1*
-rw-r-x--x 2 Remigio users        41 Oct 19 14:47 fichero2*
-rw-r--r-- 1 Remigio users        99 Oct 19 14:35 nuevo_texto
```

- Crear un fichero con plantas y otro con animales y ordenarlos juntos:

```
Remigio:~$ cat >animales
oso
nosotros)
pez
elefante
```

(esto lo escribimos)

```

rana
^D                                (pulsamos ^D)
Remigio:~$ cat >plantas
pino                               (esto lo escribimos
nosotros)
encina
rosa
alcornoque
^D                                (pulsamos ^D)
Remigio:~$ sort plantas animales
alcornoque
elefante
encina
oso
pez
pino
rana
rosa

```

basename y dirname

El sistema UNIX proporciona órdenes para construir nombres de ficheros y directorios a partir de nombres de camino completos, y viceversa. Por ejemplo, tú puedes tener una variable de entorno que contenga un nombre de camino completo, y se podría desear conocer únicamente el nombre de fichero específico.

```

$ echo $EDITOR

/usr/bin/vi

$ MIEDIT=`basename $EDITOR`

$ echo "Mi editor es $MIEDIT"

Mi editor es vi

$

```

Se usa la orden **basename** para obtener el último componente de un nombre de camino completo sobre la salida estándar. Esto generalmente proporciona el nombre de un fichero del que se ha suprimido el camino de directorio. La orden **dirname**, por otra parte, suministra sólo la sección de directorio suprimiendo el nombre de fichero.

```

$ EDDIR=`dirname $EDITOR`

$ echo "Mi editor $MIEDIT está en el directorio $EDDIR"

Mi editor vi está en el directorio /usr/bin

$

```

grep

El comando `grep` (*get regular expression and print*) sirve para mostrar las líneas de un fichero que contienen un patrón, es decir que contienen un texto. Si especificamos varios ficheros, se buscará en todos ellos; si no especificamos ninguno se buscará en la entrada estándar.

Sintaxis

grep [opciones] patrón [ficheros]

El funcionamiento normal del comando es mostrar en la salida estándar las líneas que contienen el patrón (este funcionamiento se puede alterar mediante las opciones de la **¡Error! No se encuentra el origen de la referencia.**). Además, si se encuentra el patrón alguna vez, el comando termina con éxito, y si no, termina con error.

El patrón puede ser una cadena literal o un patrón con comodines. Los comodines básicos que podemos usar son los recogidos en la tabla posterior.

Carácter	Significado
.	Cualquier carácter
[^...]	Negación del intervalo; es decir, en esta posición se permiten todos los caracteres que no se incluyen entre corchetes (conjunto complementario).
	Un sólo carácter (puede ser también un espacio en blanco)
\{...\}	Carácter de repetición. Dentro de los corchetes puede especificarse la frecuencia con que el carácter o caracteres especiales pueden aparecer. Existen tres posibilidades diferentes para definir el número: 1. \{cifra\ 2. \{cifra,\} 3. \{cifra1,cifra2\
[]	Cualquier carácter del conjunto o rango encerrado (p.e. [0-9A-Z])
^	Comienzo de línea
\$	Final de línea
<	Comienzo de palabra (\<)
>	Final de palabra (\>)

Cada comodín sólo sustituye a un carácter, aunque se pueden combinar con otros símbolos para ampliar las posibilidades, como muestra la tabla que se muestra adelante. Si queremos buscar un carácter que funciona como comodín (por ejemplo, queremos buscar un punto), podemos anteponer una barra inversa delante del carácter. De esta forma eliminamos el valor especial del comodín; pero la barra inversa es un carácter especial del shell, por lo que ésta debe aparecer entre comillas, o bien estar precedida de otra barra inversa

Carácter	Significado
x^*	x se puede repetir de 0 a n veces (x es un literal o un comodín).
$x?$	x se puede repetir 0 o 1 veces (x es un literal o un comodín).
x^+	x se puede repetir de 1 a n veces (x es un literal o un comodín).
$x\{n\}$	x se repite n veces (x es un literal o un comodín y n un número).
$x\{n, \}$	x se repite n o más veces (x es un literal o un comodín y n un número).
$x\{, n\}$	x se puede repetir de 0 a n veces (x es un literal o un comodín y n un número).
$x\{n, m\}$	x se puede repetir de n a m veces (x es un literal o un comodín y n y m números).

También podemos agrupar subexpresiones utilizando los paréntesis y utilizar la operación OR lógica con el símbolo de la tubería (`|`).

La comparación de patrones hace distinción entre mayúsculas y minúsculas a menos que empleemos la opción `-i`. Por defecto `grep` muestra las líneas que contienen el patrón, pero podemos obtener otros comportamientos utilizando las opciones adecuadas:

- Para las líneas que no contienen el patrón utilizamos `-v`.
- Para ver el número de líneas que contienen el patrón utilizamos `-c`.
- Para ver los archivos que contienen el patrón en alguna línea utilizamos `-l`.

Si la expresión que queremos buscar comienza por un guión (`-`), podría confundirse con una opción del parámetro. Entonces podemos utilizar la opción `-e` justo antes de la expresión para indicar que lo que viene a continuación no es una opción, sino la expresión. Por ejemplo, para localizar el patrón `"-v-` en los ficheros debemos escribir:

```
| grep -e -v- * |
```

Otros ejemplos:

- Buscar nombres de usuario que comiencen por `"a"`:

```
| grep "^a" /etc/passwd |
```

- Mostrar el fichero `informe` sin las líneas que contengan las palabras `"TOP SECRET"`:

```
| grep -v "TOP SECRET" informe |
```

- Mostrar los ficheros en `C` que contienen líneas vacías:

Esta tabla resume las principales opciones del comando `grep`.

Modificador	Resultado
<code>-i</code>	No hace distinción entre mayúsculas y minúsculas.
<code>-v</code>	Muestra las líneas que no contienen el patrón.
<code>-c</code>	Muestra el número de líneas que contienen el patrón.
<code>-l</code>	Muestra los archivos que contienen el patrón en alguna línea, en lugar de las líneas.
<code>-e</code>	Permite especificar que lo que viene a continuación es una expresión (y no una opción).
<code>-n</code>	Muestra el número de línea en que aparece la expresión en cada fichero.
<code>-s</code>	No muestra mensajes de error cuando algún fichero no se puede leer.
<code>-q</code>	No muestra ninguna salida; sólo termina con éxito o con error.

Existen variantes de `grep`, como por ejemplo `fgrep` y `egrep`, que tienen opciones adicionales.

find

El comando `find` sirve para buscar ficheros en el árbol de directorios. El criterio de búsqueda es muy flexible, permitiendo buscar por nombre, tamaño, fecha, etc. También podemos especificar la rama o ramas del árbol de directorios por las que buscar, y permite programar la acción a realizar con los ficheros localizados.

Sintaxis

`find` rutas [*criterios*] [*acción*]

- `rutas` es el conjunto de directorios donde comenzar a buscar. `find` buscará en las rutas indicadas y en todos sus subdirectorios.
- `criterios` es el criterio de búsqueda, y consiste en una opción que indica la forma de buscar (por nombre, por fecha, etc.) y el patrón de comparación (la máscara de nombre, la fecha, etc.). La **¡Error! No se encuentra el origen de la referencia.** recoge los posible criterios. Podemos combinar varios criterios, uniéndolos con los operadores AND y OR. Para hacerlo con AND, basta con escribir más de un criterio de búsqueda; para hacerlo

con OR, anteponeamos `-o` al criterio. Además, podemos anteponer una admiración (!) a un criterio para que funcione al contrario (busca los ficheros que no cumplan la condición). Además, podemos especificar el orden en que se evalúan las condiciones utilizando paréntesis. Puesto que los paréntesis son un carácter especial para el shell, debemos anteponer a cada paréntesis una barra invertida (\), encerrarlos entre comillas, etc.

- **acción** es la tarea que debe ejecutar `find` cada vez que localice un archivo que verifique las condiciones de búsqueda. La **¡Error! No se encuentra el origen de la referencia.** contiene las principales acciones posibles.

Si se omiten los criterios, todos los ficheros de los directorios indicados se consideran encontrados. En los criterios que comparan con números podemos anteponer al número "-" para expresar que la comparación es cierta si el número es menor que el indicado, o "+" para expresar que la comparación es cierta si el número es mayor que el indicado. (Por defecto la comparación es verdadera si el número es igual al indicado.)

Selector	Significado (criterio de búsqueda)
<code>-name nombre</code>	Busca ficheros cuyo nombre coincida con <i>nombre</i> .
<code>-group nombre</code>	Selección por pertenencia a grupo
<code>-perm modo</code>	Busca ficheros cuyos permisos coincidan con <i>modo</i> . Los permisos se especifican con el formato numérico (ver apartado ¡Error! No se encuentra el origen de la referencia. en la página ¡Error! Marcador no definido.). Si <i>modo</i> va precedido de un signo menos (-) busca ficheros que tengan al menos los permisos indicados, y si va precedido de un signo más (+), busca ficheros que tengan alguno de los permisos indicados.
<code>-type x</code>	Busca ficheros cuyo tipo sea el indicado con <i>x</i> , donde <i>x</i> puede ser: <code>f</code> para buscar ficheros normales, <code>d</code> para buscar directorios, <code>l</code> para buscar enlaces simbólicos, <code>c</code> para buscar dispositivos de caracteres, etc.
<code>-links ###</code>	Busca ficheros que tengan el número <i>###</i> de enlaces duros. Si usamos <code>-###</code> significa menos de <i>###</i> enlaces. Si usamos <code>+###</code> significa más de <i>###</i> enlaces.
<code>-size ###</code>	Busca ficheros que tengan un tamaño de <i>###</i> bloques. Para asegurarnos de que el tamaño de bloque es de un kilobyte podemos poner <i>###k</i> . Además, si usamos <code>-###</code> significa menos de <i>###</i> bloques y si usamos <code>+###</code> significa más de <i>###</i> bloques.
<code>-user nombre</code>	Busca ficheros que pertenezcan al usuario <i>nombre</i> .
<code>-atime ###</code>	Busca ficheros que hayan sido accedidos hace <i>###</i> días. Si usamos <code>-###</code> significa menos de <i>###</i> días. Si usamos <code>+###</code> significa más de <i>###</i> días.
<code>-mtime ###</code>	Busca ficheros que hayan sido modificados hace <i>###</i> días. Si usamos <code>-###</code> significa menos de <i>###</i> días. Si usamos <code>+###</code> significa más de <i>###</i> días.
<code>-ctime ###</code>	Selección por fecha de creación del archivo.
<code>-newer nombre</code>	Busca ficheros más recientes que el fichero <i>nombre</i> .

Selector	Significado (criterio de búsqueda)
<code>-print</code>	Imprimir por la salida estándar el nombre del archivo localizado.
<code>-exec <i>comando</i></code>	Ejecutar el <i>comando</i> indicado. Para indicar en el comando el nombre del fichero localizado, utilizamos {}, que se sustituirá por dicho nombre. Además, para indicar el fin del comando, debemos poner un punto y coma(;) precedido de una barra invertida porque ; es un carácter especial para el shell). Por ejemplo <code>-exec cat {} \;</code> hace que se muestre el contenido del fichero localizado.

Veamos algunos ejemplos de uso de `find`:

- Buscar ficheros que comiencen por "a" en el directorio actual y en /tmp (y subdirectorios de ambos) y mostrar dónde esta:

```
| find . /tmp -name "a*" -print |
```

- Localizar el fichero `passwd` en el disco y mostrar dónde esta:

```
| find / -name passwd -print |
```

- Mostrar el contenido de los ficheros modificados esta semana en los directorios del usuario Remigio que ocupen menos de 2 bloques de disco:

```
| find /home/Remigio -mtime -7 -size -2 -exec cat {} \; |
```

- Borrar los ficheros que tengan el nombre `*.c` y tengan más de 30 días, y también aquellos que sean del usuario *fgarcia*:

```
| find / \( -mtime +30 -name "*.c" \) -o -user fgarcia -exec rm -f {} \; |
```

NOTA IMPORTANTE: Hay que tener cuidado con el uso de máscaras en los criterios de búsqueda de nombres de fichero, ya que las máscaras se podrían sustituir por los nombres de archivos del directorio actual que encajen en las máscaras, de manera que se buscarían esos nombres de ficheros del directorio actual, en lugar de los nombres de fichero que indiquen las máscaras. Para evitar el problema, encerramos entre comillas las máscaras (como en los ejemplos anteriores), de manera que la máscara llega tal cual al comando, y no es interpretada por el shell antes de ser entregada al comando.

Este problema puede observarse también en otros comandos, además de en `find`.

egrep

Este comando busca la aparición de una plantilla de búsqueda en un archivo y amplía las posibilidades del comando **Grez**. En la plantilla de búsqueda del comando **egrep** pueden utilizarse adicionalmente caracteres especiales.

Sintaxis

Egrep [opciones] [plantilla_de_búsqueda] [archivos]

El comando **egrep** admite también otra opción de gran utilidad: si se incorpora al comando **egrep** la opción **-f** y otro nombre de archivo, lee automáticamente la plantilla de búsqueda del archivo y trata entonces con esta plantilla de búsqueda el archivo deseado.

```
Egrep -f archivoplantilla archivobusqueda
```

En este caso, el comando **egrep** lee en primer lugar el archivo *ArchivoPlantilla* y extrae de ahí la plantilla de búsqueda que utiliza con el archivo *ArchivoBúsqueda*. Se admiten las siguientes opciones:

Comando	Significado
-v	Se muestran todas las líneas excepto la que contiene la plantilla de búsqueda.
-c	Se indica sólo el número de líneas encontradas.
-i	En la búsqueda no debe haber distinción entre mayúsculas y minúsculas.
-l	En líneas individuales aparecen sólo los nombres de los archivos en los que se ha encontrado la plantilla de búsqueda.
-n	Delante de la línea aparece el número que le corresponde.
-e RA	Como plantilla de búsqueda debe usarse RA. Es útil siempre que la plantilla de búsqueda empiece por un signo de menos.

Junto a los caracteres especiales que ya admite el comando **Grez**, también pueden usarse adicionalmente los siguientes:

Carácter	Significado
+	Carácter de repetición. El carácter o carácter especial introducido delante de este signo debe aparecer como mínimo una vez, a diferencia de lo que sucede con el carácter "*", en el que el carácter podría también no aparecer.
?	Carácter de repetición. El carácter o carácter especial que aparece ante éste puede aparecer una vez o no aparecer.
	A la izquierda y derecha de este signo aparecen plantillas. Ambas expresiones se permiten en esta posición. Una conjunción disyuntiva de plantillas.
(...)	Con los paréntesis pueden agruparse secciones de la plantilla.

fgrep

De forma similar a los comandos **egrep** y **Grez**, este comando busca el texto de búsqueda en los archivos indicados. Sin embargo, a diferencia de los dos comandos anteriores, en este caso el texto de búsqueda no debe contener ningún carácter especial necesario para la creación de expresiones regulares. Si no aparece ningún archivo como último parámetro, el comando **fgrep** lee a través del canal de entrada estándar.

Estas son las opciones del comando **fgrep**:

Comando	Significado
-v	Se muestran todas las líneas excepto la que contiene la plantilla de búsqueda.
-c	Se indica sólo el número de líneas encontradas.
-i	En la búsqueda no debe haber distinción entre mayúsculas y minúsculas.
-l	En líneas individuales aparecen sólo los nombres de los archivos en los que se ha encontrado la plantilla de búsqueda.
-n	Delante de la línea aparece el número que le corresponde.
-x	Sólo se mostrarán las líneas que contengan la plantilla de búsqueda de forma exacta.
-f archivo	En el archivo indicado aparece el texto a buscar. Si el archivo se compone de varios textos de búsqueda en líneas independientes, corresponde a una conjunción disyuntiva entre los textos de búsqueda, es decir, en la línea debe aparecer sólo uno de los textos de búsqueda.

Si se incorpora la opción **-f** con un nombre de archivo, no aparece otro texto de búsqueda como parámetro al ejecutar el programa.

uniq

Esta orden informa de las líneas repetidas de un fichero, pero su entrada debe estar correctamente ordenada antes de que pueda funcionar.

La orden **uniq** lee su entrada estándar o una lista de nombres de ficheros, y escribe en su salida estándar. Por omisión, **uniq** escribirá una instancia de cada línea de entrada que sea diferente a las otras líneas.

Sintaxis

uniq [opciones] [entrada [salida]]

Sólo se escribe una copia de cada línea diferente; **uniq** produce una lista de las líneas únicas del fichero. De nuevo, la entrada a **uniq** debe estar en orden correcto o si no **uniq** fallará.

Esta orden **uniq** suele usarse junto a la orden **sort**. Parece lógico, ya que **uniq**, como hemos dicho, sólo puede usarse sobre ficheros previamente ordenados.

Modificador	Resultado
-u	Hace que sólo se envíen a la salida las líneas no repetidas.
-d	Hace que sólo se envíen a la salida las líneas repetidas (sólo envía una copia).
-c	Muestra a la izquierda de cada línea el número de veces que se repite.
-n	Se ignoran los primeros <i>n</i> campos de la línea en su comparación
+n	Se ignorarán los <i>n</i> primeros caracteres de la línea.

Cuando **-n** y **+n** se usan juntos, se omiten los campos antes que los caracteres.

comm

El comando **comm** sirve para separar las líneas comunes y no comunes de dos ficheros (**fichero1** y **fichero2**) previamente ordenados, y genera en la salida estándar un informe con 3 columnas: líneas que sólo están en **fichero1**, líneas que sólo están en **fichero2** y líneas que están en los dos ficheros.

<i>Sintaxis</i>
comm [<i>opciones</i>] fichero1 fichero2

Las opciones **-1**, **-2** y **-3** hacen que la columna correspondiente no se muestre, como se indica en la tabla adjunta.

Modificador	Resultado
-1	Hace que no se muestre la columna primera (líneas que sólo están en el primer fichero).
-2	Hace que no se muestre la columna segunda (líneas que sólo están en el segundo fichero).
-3	Hace que no se muestre la columna tercera (líneas que están en ambos ficheros).

cmp

El comando `cmp` (*compare*) sirve para comparar dos ficheros, no necesariamente ordenados. Como salida, imprime un mensaje que contiene el número de byte (carácter) dentro del fichero y el número de línea donde aparece la primera diferencia entre los ficheros comparados (las líneas y caracteres se numeran comenzando en 1). Si los ficheros son iguales, no imprime nada.

Sintaxis

cmp [*opciones*] fichero1 fichero2

`cmp` se detiene ante la primera diferencia. Con la opción `-l` podemos visualizar todas las diferencias en tres columnas que contienen: posición de la primera diferencia (byte), código ASCII en octal del carácter que aparece en el primer fichero en esa posición, y código ASCII en octal del carácter que aparece en el segundo fichero en esa posición. La siguiente tabla contiene otros modificadores que podemos utilizar con `cmp`.

Modificador	Resultado
<code>-l</code>	Imprime la lista completa de diferencias indicando en 3 columnas la posición y los caracteres que difieren.
<code>-s</code>	No imprime nada cuando los ficheros son diferentes, sino que termina con error.

Un valor de retorno 0 significa que los ficheros son iguales; una valor de retorno 1 significa que son diferentes. El resto de valores de retorno indican errores.

diff

Esta es una orden muy potente usada para ficheros de texto. Esta orden produce un índice completo de todas las líneas que difieren entre dos archivos, junto con sus números de líneas y lo que debe ser modificado para hacer iguales los ficheros.

Sintaxis

diff [*opciones*] archivo1 archivo2

Las diferencias pueden igualarse de tres formas distintas:

- Introduciendo algo a las líneas del primer archivo (“a”).
- Suprimiendo líneas del primer archivo (“d”).
- Sustituyendo líneas del primero por líneas del segundo (“c”).

Para la descripción de las diferencias se usan abreviaturas. Los valores delante de **n1** y **n2** son números de líneas del primer archivo; los valores delante de **n3** y **n4** son números de líneas del segundo archivo:

`n1 a n3,n4`

`n1,n2 d n3`

`n1,n2 c n3,n4`

Tras una línea en este formato aparece en la salida el contenido de las líneas correspondientes.

Delante de las líneas del primer archivo aparece el carácter “<”; delante de las líneas del segundo archivo, el carácter “>”.

Las siguientes opciones con las que están permitidas con el uso de **diff**.

Modificador	Resultado
-b	Indica que los espacios en blanco son irrelevantes para la comparación del contenido de las líneas.
-e	Con la ayuda de esta opción pueden crearse los scripts <i>ed</i> , con los que puede realizarse la concordancia de ambos archivos.
-f	Esta opción invierte la función de la opción <i>-e</i> . Es decir, se crean scripts <i>ed</i> que permiten adaptar el segundo archivo al primero.

El valor de retorno del comando **diff** muestra si el comando pudo llevar a cabo su labor de forma exhaustiva. Si ambos archivos son idénticos, aparece el valor de retorno 0. Si existen diferencias, el valor de retorno es el 1, y, finalmente, el valor de retorno 2 aparece cuando surge algún error

El comando **diff** puede introducirse con facilidad en el interior de scripts de shell en los que es importante saber si dos archivos son diferentes.

Si el comando **diff** debe leer los datos de alguno de los archivos a través del canal de entrada estándar, este hecho se especificará mediante el nombre de archivo “-”.

Si uno de los dos parámetros corresponde al nombre de un directorio, en ese directorio se buscará un archivo con el mismo nombre que el otro archivo.

pg

El programa **pg** presenta como salida una pantalla cada vez, esperando a que presionemos <INTRO> antes de continuar con la siguiente pantalla. Generalmente **pg** se usa al final de una línea de cauce o con una lista de nombres de ficheros como argumentos.

La orden **pg** hará una pausa después de cada pantalla completa de salida, indicando con “:” (dos puntos) el final de la pantalla. Si introducimos un número antes de presionar <INTRO>, **pg** saltará hacia delante ese número de pantalla antes de visualizar otra. Si el número es negativo, **pg** saltará hacia atrás ese número de pantallas completas. El parámetro **l** hace que **pg** se mueva hacia delante línea a línea en lugar de pantalla a pantalla. Puede también tomar un argumento positivo o negativo para especificar cuántas líneas pasar.

Además, **pg** puede aceptar muchas otras órdenes tras el inductor “:”. Podemos introducir una expresión regular a continuación de : con la sintaxis de **ed**, comenzando con / o ?, y **pg** buscará hacia delante o hacia atrás, respectivamente, esa expresión. Luego visualizará el segmento de texto que circunda a la expresión coincidente. Un número *n* precediendo a / hará que **pg** busque la *enésima* ocurrencia de la expresión:

```
:6/[fe]grep/
```

Este ejemplo buscará la sexta ocurrencia de la cadena **fgrep** o **egrep** a continuación de la línea actual. El operador . (punto) hace que la pantalla actual sea revisualizada, y **\$** hace que se visualice la última pantalla completa del fichero.

Si hay más de un fichero en la lista de argumentos de **pg**, **n** saltará al comienzo del siguiente fichero, y **p** saltará hacia atrás al comienzo del fichero anterior. Ambos **n** y **p** pueden tomar un número antes de la letra para especificar el *enésimo* fichero anterior:

```
:3p
```

Naturalmente, las órdenes **n** y **p** no funcionarán si **pg** se invoca al final de una línea de cauce. La orden **h** mostrará una lista de las órdenes de **pg**, y **q** o la tecla SUPR harán que **pg** termine, regresando al shell.

La orden **pg** usa la variable de entorno TERM para formatear la salida en la mayoría de los terminales de pantalla total, de modo que esta variable debe ser correcta o **pg** actuará de forma confusa.

La orden **pg** comienza su visualización al principio del fichero. Para ver el fin de un fichero con **pg** debemos ir paso a paso a través del fichero completo o bien usar el operador **\$**.

nl

Con esta orden podemos numerar las líneas de un archivo pasado como parámetro desde la línea de órdenes.

Sintaxis

nl archivo

Una opción de uso muy recomendable es la de la utilización de **nl** junto con la orden **sort** como entrada, y mediante un redireccionamiento. Así,

```
sort fichero1 | nl
```

6.9.- Guardar y acceder a los nombres de directorios más usados.

Existen algunos directorios a los que se vuelve con más frecuencia. Para facilitar el acceso a estos directorios se puede crear y utilizar sus propias variables *shell* temporales.

Las variables *shell* temporales deben ir en mayúsculas, para evitar confusión con las palabras en minúsculas reservadas para UNIX. Deben ser cortas para que sean de fácil escritura. Seguiremos los siguientes pasos:

- Usaremos **cd** para un directorio que tendrá que volver una y otra vez.
- Escribimos la variable, del modo **DI='pwd'**, donde DI es el nombre que habremos otorgado a la variable.
- Nos aseguraremos de escribir las comillas inversas, y no los apóstrofes. No debemos dejar ningún espacio en blanco. El comando guardará la ruta de acceso completa del directorio actual en la variable *shell* DI. Para volver al directorio actual usando DI, utilizaremos **cd \$DI**. El signo **\$** se usa para acceder al contenido de la variable, pero no para fijarlo (es decir, para otorgarle ese valor).
- Si seguimos los pasos anteriores, podemos hacer tantas variables como queramos guardar, y cambiar, fácilmente, entre varios directorios.
- Tenemos que estar en el directorio deseado antes de realizar los pasos anteriores.

- No podemos elegir el nombre de una variable shell que ya exista.
- Para ver una lista con las variables existentes, debemos escribir **env**. Estas variables duran hasta que se salga de la sesión. Si queremos que algunas de estas variables sean permanentes, debemos configurarlas en *.profile*.

6.10.- Diferencias más destacadas entre MS-DOS y UNIX

* Directorio raíz. Para designar el directorio raíz y separar el nombre de los directorios, MS-DOS usa el símbolo \, mientras que UNIX usa /.

* Comando cd. Este comando es similar. Sus diferencias son:

- UNIX requiere un espacio entre **cd** y el nombre del directorio (incluido ..).
- UNIX no permite poner el nombre entero 'chdir'.
- En UNIX, **cd** sin un directorio nos lleva a **/home**.

Manipulación de archivos

En este capítulo abordaremos las operaciones más importantes que pueden realizarse con los archivos. En el anterior ya vimos la creación, el borrado, visualización, etc. Aquí afrontaremos el tema de la copia de ficheros de dispositivos, los enlaces a ficheros y temas relacionados

7.1.- Uso de los metacaracteres

Las órdenes vistas en el capítulo anterior, de manipulación de archivos y directorios, admiten el uso de ‘metacaracteres’, pero deben usarse con bastante precaución. Los metacaracteres los vimos en capítulos anteriores.

7.2.- ¿Cómo abortar el desarrollo de una orden?

Para abortar el desarrollo de una acción desencadenada por una orden se pulsa simultáneamente [**ctrl.** + **pausa**]. Esto puede ser preciso en la salida por pantalla tras una orden **cat**, **more** o **pg**. En el caso de **pg** y **more** se puede pulsar **q** cuando aparecen los dos puntos (:) o el mensaje *more n%*.

7.3.- Enlaces

Son un tipo de archivos que permiten que un mismo archivo o directorio tenga varios nombres o ubicaciones en el árbol, aunque una sola ubicación física.

7.3.1.- Enlaces duros

UNIX hace referencia a todos los archivos y directorios con un número de i-nodo, o nodo-i. Éste es un número de entrada en la tabla de i-nodos. Cada sistema de archivos tiene su propia tabla de i-nodos, y cada entrada de i-nodo en uso contiene información sobre un archivo en ese sistema de archivos.

Una entrada de i-nodo contiene la mayoría de la información sobre un archivo, como es el propietario; el grupo; los permisos; tipo de archivo; fecha y hora del último acceso; fecha y hora de la última modificación; etc. Lo que no contiene es el nombre del archivo.

UNIX admite que más de un nombre de archivo haga referencia a un número de i-nodo dado, lo que permite múltiples nombres para el mismo archivo. Como el nombre de archivo no forma parte de la información de i-nodo, cada uno de los nombres de archivo es válido, sin importar cuál fue el primer nombre.

Cuando se tiene varios nombres para el mismo archivo, a cada nombre se le llama **enlace duro** al archivo. Esto permite poner el mismo archivo en directorios diferentes. L

La entrada de directorio (el nombre del enlace) referencia a la zona del disco en la que se encuentran los datos del archivo enlazado. En esa zona del disco, además de los datos se almacena el número de enlaces que tiene ese archivo. El fichero original es también un enlace. Así, al borrar un enlace (aunque sea el del fichero original) se quita la entrada del directorio (el nombre del enlace), pero los datos contenidos no se eliminan del disco. En su lugar se decrementa el número de enlaces restantes y, si no quedan más enlaces, entonces se elimina el contenido del fichero. Cada vez que se añade un enlace para ese fichero, se incrementa el número de enlaces.

7.3.2.- Enlaces blandos o *simbólicos*

La entrada de directorio (el nombre del enlace) es fichero nuevo que contiene datos (ocupa espacio en disco), y esos datos son una cadena de texto que referencia al fichero original. El fichero original no funciona como otro enlace simbólico. Si borramos o renombramos o movemos el fichero original, los enlaces simbólicos quedarán inconsistentes, apuntando a un objeto que ya no existe. (Se parece a los accesos directos de Windows).

El comando `ln` admite como parámetro los nombres del fichero original y del enlace:

Sintaxis

ln [*opciones*] origen destino

Como origen especificamos uno o varios ficheros o directorios, o una máscara. Como destino especificamos el nombre del enlace (si el origen es un único archivo o directorio), o bien un directorio donde situar los enlaces creados (en este caso el origen son varios archivos o directorios, o una máscara, y los enlaces se crean con los mismos nombres que el origen, en el directorio de destino). Si omitimos el destino, se supone que es el directorio actual, por lo que el origen debe encontrarse en un directorio diferente del actual.

El principal modificador que podemos utilizar es `-s`, que sirve para que se creen enlaces simbólicos en lugar de enlaces duros.

Por defecto, `ln` no sobrescribe los nombres de ficheros que existan para crear, en su lugar, enlaces. Necesitamos utilizar la opción `-f` para que esto ocurra. Además, con la opción `-i` nos preguntará antes de sobrescribir algún fichero existente.

En circunstancias normales los usuarios no podemos crear enlaces duros a directorios (el sistema sí puede hacerlo). Para directorios sólo podemos hacer

enlaces simbólicos. Estos enlaces a directorios se manejan siempre como si fueran realmente directorios, excepto por que para eliminarlos no debemos utilizar `rmdir`, sino `rm`, ya que realmente no son directorios sino un tipo especial de ficheros.

Para identificar los enlaces simbólicos podemos utilizar (entre otras muchas cosas) el comando `ls` con la opción `-l`, ya que junto con los permisos de los ficheros aparece el carácter que identifica el tipo de fichero, que en el caso de los enlaces es `"l"`. Por ejemplo: supongamos que en un directorio existen dos entradas: el directorio `"dir1"` y un enlace simbólico a éste, llamado `"otro_dir1"`; al hacer `ls -l` observamos que `dir1` es de tipo directorio, y `otro_dir1` es un enlace que apunta, precisamente, a `dir1` (esto lo vemos junto al nombre, a la derecha de la línea):

```
Remigio:~$ mkdir dir1
Remigio:~$ ln -s dir1 otro_dir1
Remigio:~$ ls -l
total 1
drwxr-xr-x  2 Remigio users      1024 Oct 15 09:59 dir1/
lrwxrwxrwx  1 Remigio users      4 Oct 15 09:59 otro_dir1 -> dir1/
```

Los enlaces duros no se pueden identificar, ya que son realmente del mismo tipo que sus originales, y no hay manera de distinguirlos de éstos (son idénticos a todos los efectos). Lo que sí podemos es averiguar cuántos enlaces recaen sobre la misma zona de disco (cuántos enlaces apuntan a un fichero). También lo podemos hacer con `ls -l`, ya que la segunda columna indica el número de enlaces que se mantiene para un fichero. En el siguiente ejemplo utilizamos también la opción `-a` para que se muestren también los directorios `"."` y `".."`:

```
Remigio:~$ ls -al
total 7
drwx--x--x  3 Remigio users      1024 Oct 15 10:08 ./
drwxr-xr-x 16 root      root      5120 Oct 14 11:27 ../
drwxr-xr-x  2 Remigio users      1024 Oct 15 10:01 dir1/
lrwxrwxrwx  1 Remigio users      4 Oct 15 09:59 otro_dir1 -> dir1/
```

Observamos que:

- El enlace `otro_dir1` sólo tiene 1 enlace, ya que nadie más enlaza con él.
- El directorio `dir1` tiene dos enlaces: el suyo propio, y el directorio `"dir1/."`, que es un enlace duro a `dir1` creado por el sistema cuando se creó (p.e. con `mkdir`) el directorio `dir1` (¡no se cuentan los enlaces simbólicos!).
- El directorio `"."` tiene 3 enlaces: el propio `"."`, el de `"/home/Remigio"` y el de `"/home/Remigio/dir1/.."`.
- El directorio `".."` tiene múltiples enlaces: el propio `".."` (su ruta completa es `"/home/Remigio/.."`), el de `"/home"`, el de `"/home/."` y el precedente del `".."` de otros 13 directorios home de otros usuarios (de la forma `"/home/USUARIO/.."`).

Los distintos enlaces duros a un mismo archivo comparten la siguiente información (aparte de los datos contenidos en fichero, su tamaño, etc.):

- El tipo de archivo.
- El propietario.
- El grupo.
- Los permisos.
- La fecha de modificación, acceso, etc.

La tabla que sigue resume las principales opciones que admite el comando `ln`. Muchas de ellas son semejantes a las utilizadas en otros comandos.

Modificador	Resultado
<code>-s</code>	Indica que los enlaces que se deben crear serán simbólicos (y no duros)
<code>-f</code>	Indica que se sobrescriban los ficheros existentes en el destino si es necesario.
<code>-i</code>	Interactivo: pregunta antes de sobrescribir un fichero existente en el directorio de destino.
<code>-b</code>	Crea una copia de seguridad de los archivos que sobrescribe.
<code>-v</code>	Muestra el nombre de cada fichero antes de crear enlace.

Algunos ejemplos de uso de `ln`:

- Crear un enlace simbólico de todos los dispositivos de tipo `tty` en un subdirectorio llamado `ttys`.
- Crear un enlace al directorio de dispositivos llamado `acceso_a_dispositivos`.
- Hacer un enlace de todos los ficheros que tenga el usuario `fjgarcia` en su directorio `practicas` a nuestro directorio `practica_copiada`, sobrescribiendo los ficheros que pudiera haber en éste directorio.
- Hacer un enlace del fichero `/home/profesor/notas_practicas` al fichero `mi_suspenso`.

```
Remigio:~$ ln -s /dev/tty* ttys
Remigio:~$ ln -s /dev/tty acceso_a_dispositivos
Remigio:~$ ln -f /home/fjgarcia/practica/* ~/practica_copiada
Remigio:~$ ln /home/profesor/notas_practicas mi_suspenso
```

Debemos prestar especial atención cuando construyamos enlaces simbólicos entre objetos ubicados en directorios diferentes utilizando rutas relativas, ya que el enlace buscará el objeto original siguiendo la ruta especificada en el momento de la creación, pero interpretándola como relativa a la ubicación actual del enlace. Veamos este problema con un ejemplo.

Supongamos que tenemos dos directorios: `dir1` y `dir2`. Dentro de `dir1`, tenemos el fichero `mi_fichero`, y ahora queremos crear un enlace simbólico al fichero, desde el directorio `dir2`:

```
Remigio:~$ mkdir dir1
Remigio:~$ mkdir dir2
Remigio:~$ cat >dir1/mi_fichero
```

```
Este es el contenido de mi_fichero
^D
```

El siguiente intento de crear el enlace es incorrecto. A pesar de que `ln` crea un enlace (al hacer `ls -l` aparece el enlace con la ruta indicada al crearlo), no podemos ver el contenido del fichero original:

```
Remigio:~$ ln -s dir1/mi_fichero dir2
Remigio:~$ cd dir2
Remigio:~/dir2$ ls -l
total 0
lrwxrwxrwx  1 Remigio users      15 Jan 25 11:14 mi_fichero ->
dir1/mi_fichero
Remigio:~/dir2$ cat mi_fichero
cat: mi_fichero: No such file or directory
```

La razón de que sea incorrecto, es que el enlace busca el archivo original como "`dir1/mi_fichero`" desde su ubicación actual (`~/dir2/`), es decir, que interpreta que el original es "`~/dir2/dir1/mi_fichero`". Donde en realidad se encuentra el original (utilizando una ruta relativa desde el directorio donde se ubica el enlace) es en "`../dir1/mi_fichero`". Luego esta es la ruta que deberíamos haber empleado al crear el enlace, a pesar de que, dado el directorio en el que nos encontrábamos en el momento de crear el enlace, esta última ruta no conduce al original (lo que ocurre es que sí es correcta desde la ubicación final del enlace). En consecuencia, la forma correcta de crear el enlace es la siguiente:

```
Remigio:~$ ln -s ../dir1/mi_fichero dir2
Remigio:~$ cd dir2
Remigio:~/dir2$ ls -l
total 0
lrwxrwxrwx  1 Remigio users      18 Jan 25 11:24 mi_fichero ->
../dir1/mi_fichero
Remigio:~/dir2$ cat mi_fichero
Este es el contenido de mi_fichero
Remigio:~/dir2$
```

Recordemos que este problema sólo se plantea cuando trabajamos con enlaces simbólicos; para los enlaces duros la ruta de enlace que debemos especificar es la que sea correcta en el momento de la creación.

La ruta del enlace simbólico hacia su destino no se modifica al cambiar de ubicación el enlace. Siguiendo con el ejemplo, si cambiamos el enlace `mi_fichero` a otra ubicación (por ejemplo un directorio `dir3` dentro de `dir2`), el enlace dejará de ser correcto:

```
Remigio:~/dir2$ mkdir dir3
Remigio:~/dir2$ mv mi_fichero dir3/
Remigio:~/dir2$ cd dir3/
Remigio:~/dir2/dir3$ ls -l
total 0
lrwxrwxrwx  1 Remigio users      18 Jan 25 11:24 mi_fichero ->
../dir1/mi_fichero
Remigio:~/dir2/dir3$ cat mi_fichero
```



```
cat: mi_fichero: No such file or directory
```

En cualquier caso, para evitar posibles problemas, es mejor utilizar rutas absolutas. Por ejemplo:

```
Remigio:~$ ln -s ~/dir1/mi_fichero dir2
Remigio:~$ cd dir2
Remigio:~/dir2$ ls -l
total 0
lrwxrwxrwx  1 Remigio users    29 Jan 25 11:45 mi_fichero ->
/home/Remigio/dir1/mi_fichero
Remigio:~/dir2$ cat mi_fichero
Este es el contenido de mi_fichero
```

7.3.3.- Notas acerca de los enlaces *simbólicos*

Hay algunas órdenes que se ejecutan directamente sobre el contenido del archivo, aunque las ejecutemos sobre el enlace simbólico. Una de estas órdenes es **cp**.

Sin embargo, hay otras (las que manejan el nombre de los archivos, y no su contenido), que se ejecutan sobre el enlace en sí. Por ejemplo, la orden **mv** (ver capítulo anterior para saber su función).

No debemos enlazar nunca un directorio a sí mismo, ya que se podría crear un bucle infinito.

7.4.- Borrar un enlace duro

Si un archivo tiene múltiples enlaces duros significa que tiene múltiples nombres. Si se usa **rm** para borrar uno de esos nombres, el resto de nombres y el contenido del archivo no se verán afectados, pero el número de enlaces actual para ese archivo se ve reducido en uno. Si se borra un nombre de archivo que presenta una cuenta de enlaces de 1, el número de enlaces será 0, lo que significa que no se ha dejado ningún nombre de acceso a ese archivo; es decir, ese archivo se ha borrado y no hay forma de recuperarlo.

7.5.- El comando *cpio*

Esta orden se usa para facilitar el acceso en bruto al disquete sin necesidad de montarlo. Se trata realmente de un programa de archivo que copia una lista de ficheros en un único fichero de salida grande, creando cabeceras de separación entre los ficheros de modo que se puedan recuperar individualmente.

Sintaxis

cpio [-o | [opciones]

cpio [-f] [opciones] [modelo]

cpio [-p] [<i>opciones</i>] directorio

Los archivos **cpio** pueden ocupar múltiples disquetes, permitiendo eficientes copias de seguridad de grandes jerarquías de directorios. Además, **cpio** preserva la propiedad y los tiempos de modificación del fichero, y puede archivar tanto texto como ficheros binarios. El uso de **cpio** es el modo más eficaz de almacenar ficheros sobre un disco flexible, y un archivo **cpio** es generalmente algo más pequeño que el conjunto de los ficheros originales que compone el archivo. El software del sistema y la mayoría de los discos instalables están generalmente en formato **cpio**.

7.5.1.- Uso de medios magnéticos para operaciones *cpio*

Antes de usar **cpio** con un disco flexible, debemos formatearlo. Sin embargo, no necesitamos **mkfs**, ya que el **cpio** sobrescribirá el sistema de ficheros cuando cree su archivo sobre el disco. De hecho, una vez que hayamos usado un disquete con **cpio**, necesitaremos crear un nuevo sistema de ficheros sobre él antes de que se pueda montar. Los discos **cpio** no son compatibles con los disquetes de sistemas de ficheros, por lo que la creación de un sistema de ficheros con **mkfs** destruirá cualquier dato almacenado en formato **cpio**.

Debemos usar el fichero de dispositivo en bruto para las operaciones **cpio**, ya que **cpio** no usa el sistema de ficheros. Puesto que **cpio** envía su archivo a su salida estándar, deberíamos usar una orden de esta forma cuando escribamos en un disco flexible:

```
$ echo nombre_fichero | cpio -o > /dev/rdisk/f0q15dt
```

Antes de ejecutar ésta o cualquier otra línea de orden **cpio**, debemos asegurarnos de tener un disco flexible formateado en la unidad correcta. El disquete debe estar capacitado para escritura.

El comando **cpio** con la opción **-o** lee el nombre del archivo a través del canal de entrada estándar y reproduce el contenido del archivo, incluida la información de gestión, por el canal estándar.

El comando **cpio** con la opción **-i** lee el contenido del archivo y los datos de gestión por el canal de entrada estándar y guarda los archivos en el disco. Pueden seleccionarse archivos determinados con ayuda de uno o varios modelos de búsqueda. Como modelo de búsqueda se permite usar todos los nombres, incluidos los caracteres especiales para la generación de nombres de archivo.

A los archivos guardados se les otorgan los mismos derechos de acceso que tienen en el medio de almacenamiento. Como propietario o como pertenencia a grupo se introducen aquellos valores que tiene el usuario que ha iniciado el comando **cpio**.

El comando **cpio** con la opción **-p** leerá los nombres del archivo a través del canal de entrada estándar y se copiarán los archivos en otra ubicación del árbol

de directorios. El nombre de directorio que aparece como parámetro indica el destino en el que deben copiarse los archivos.

El comando **cpio** admite las siguientes opciones:

Modificador	Resultado
-B	El comando cpio trabajará con bloques voluminosos, cada uno de 5120 bytes.
-c	Los datos de gestión para cada archivo deben convertirse en caracteres ASCII. De esta forma, los almacenamientos con el comando cpio serán más portables.
-t	Junto con la opción -i , se reproduce el contenido de un directorio.
-v	Muestra el nombre de los archivos copiados
-d	Al leerlos, si es necesario, se crearán los directorios de nuevo.
-u	Los archivos deben copiarse necesariamente. A continuación, se devolverán los archivos anteriores a un archivo con el mismo nombre.
-f	Deben copiarse todos los archivos que no se han encontrado con el modelo de búsqueda.
-l	Con la copia transferida de datos (opción -p) antes deben crearse archivos mediante un vínculo.

Frecuentemente **cpio** se empareja con la orden **find** para generar archivos. La orden **find** buscará en un directorio los ficheros que cumplan sus argumentos de la línea de orden y escribirá los nombres de camino en su salida estándar. Esta salida se conecta mediante cauce a **cpio** para componer el archivo.

```
$ cd

$ find . -print | cpio -oc >/tmp/cpio.propio

113 blocks

$
```

La orden **cpio** informa sobre el número de bloques escritos si no hay errores. La orden del ejemplo anterior crea un archivo que contiene todos los ficheros y subdirectorios de nuestro directorio **/home** y deja el archivo en **/tmp/cpio.propio**.

Naturalmente, podemos variar la parte **find** de la orden para modificar la lista de nombres de fichero a incluir en el archivo. Por ejemplo, para crear un fichero **cpio** sólo con los ficheros que hayan cambiado durante la última semana, usaremos la orden siguiente:

```
$ find . -mtime -7 -print | cpio -ocv > /tmp/cpio.propio
```

Debemos asegurarnos de incluir **-print** en la línea de orden **find**, o no se generarán nombres para que **cpio** los archive.

Podemos usar **cpio** para archivar un fichero que esté ya en un archivo **cpio**, y **cpio** puede seguir la pista de los niveles correctamente. Sin embargo, no

debes crear un archivo mediante la redirección de la salida a un fichero que se halle en el mismo directorio que estés archivando. Eso crearía un bucle infinito y un archivo muy grande. La orden siguiente está permitida, pero es una seria equivocación:

```
$ find . -print \ cpio -oc > ./arch.cpio
```

Finalmente, podemos hacer que **cpio** recargue solamente un subconjunto de los ficheros que haya en el archivo. Después de las opciones en la línea de orden, podemos dar un patrón usando el formato de operadores comodines del shell. El programa **cpio** hallará todos los ficheros cuyos nombres coincidan con el patrón y recarga únicamente esos ficheros. Deberíamos entrecomillar los patrones para impedir que el shell los expanda antes de que **cpio** los vea:

```
$ cpio -icvB "*fich" < /dev/fd0
```

Esta orden recuperará del archivo contenido en el disco flexible todos los ficheros cuyos nombres finalicen con la cadena **fich**.

Se permiten múltiples patrones, pero debemos rodear cada uno de ellos con dobles comillas.

```
$ cpio -icvB "*fich" "[0-9]hola*" </dev/fd0
```

Además de los ficheros con nombres de camino finalizados en **fich**, este ejemplo recuperará todos los ficheros con un dígito de 0 a 9 seguido de “hola” en cualquier lugar del nombre de camino.

La orden **cpio** también nos permite copiar una lista de ficheros desde una posición en nuestro sistema de ficheros a otra.

7.6.- El comando *tar*

Con este comando pueden guardarse archivos individuales y jerarquías completas de directorios. Los archivos y directorios a guardar se especificarán en la línea de entrada.

Sintaxis

tar [*opciones*] archivos | directorios

Si se usa – (menos) como nombre de archivo, **tar** usará la E/S estándar, por lo que se permite redirección.

Al igual que **cpio**, **tar** almacena el nombre de camino usado cuando se creó el archivo. Por tanto, deberíamos tener cuidado cuando usemos nombres de camino absoluto, ya que al extraer ficheros del archivo, podemos sobrescribir

incorrectamente ficheros existentes. Generalmente es mejor usar sólo nombres de camino relativos en las líneas de orden de **tar**.

Si el comando **tar** reconoce en sus parámetros el nombre de un directorio, se guardará de forma recursiva el directorio junto a todos los subdirectorios y archivos.

Pueden usarse las siguientes opciones:

Modificador	Resultado
-c	Crea una nueva copia de seguridad del archivo e inicializa el medio de almacenamiento de datos.
-r	Adjunta los datos a un componente de datos ya existente en el medio de almacenamiento.
-t	Visualización de todos los datos guardados.
-u	Sustitución de todos los archivos anteriores en el disco por la nueva versión de los archivos a guardar.
-x	Extrae los datos guardados en el medio de almacenamiento.
-f	Usaremos esta opción cuando no se esté escribiendo sobre un dispositivo de cinta.

Redireccionamientos. Variables de entorno.

El redireccionamiento de entrada y salida es una sencilla y a la vez sorprendente propiedad del shell. UNIX se conoce como un sistema operativo flexible en gran parte gracias a las posibilidades que directa o indirectamente nos ofrece el redireccionamiento de entrada salida.

8.1.- FUNCIONAMIENTO DEL REDIRECCIONAMIENTO

Muchos programas auxiliares disponibles en UNIX proporcionan información en pantalla, pero no siempre es posible ver toda la información de inmediato. Por ello, el sistema operativo encauza esa información hacia un archivo, que podemos leer o modificar con un editor.

Todos los comandos usan los canales de entrada y salida para reproducir o leer sus datos. El canal de entrada, normalmente usado para la lectura, está vinculado al teclado. Por lo tanto, UNIX ejecuta un canal de entrada y salida distinto para cada usuario. Además, vincula el teclado de cada usuario con el canal de entrada adecuado. El canal de salida también se administra de una forma similar. El canal de salida estándar está vinculado a la pantalla de cada usuario.

El shell puede manipular brevemente estos canales de entrada y salida para un comando introducido por el usuario, de forma que dejen de estar vinculados al teclado o a la pantalla y se vinculen a un archivo.

Entonces, los comandos leen sus datos directamente del archivo o escriben sus datos en el archivo. En realidad, los comandos no “saben” si reproducen su salida en un archivo o en pantalla.

8.2.- REDIRECCIÓN DE E/S Y TUBERÍAS

Cuando ejecutamos un comando en *bash*, por defecto, el comando utilizará como entradas y salidas estándar los mismos dispositivos que *bash*, es decir, la consola (pantalla y teclado). Sin embargo, podemos hacer que estos dispositivos sean otros, por ejemplo, para que el resultado de un comando, en lugar de mostrarse en pantalla, se almacene en un fichero, se envíe a la impresora, se descarte, etc. Esto se denomina **redirección** de entradas y salidas, y para hacerlo debemos indicar, junto con el comando, el conjunto de redirecciones necesarias.

- Para **redireccionar la entrada estándar** utilizamos el carácter menor que (<) seguido del nombre de fichero o dispositivo de donde se tomará la información de entrada.
- Para **redireccionar la salida estándar** utilizamos el carácter mayor que (>) seguido del nombre de fichero o dispositivo a donde irá a parar la información de salida. Tenemos dos posibilidades: crear un nuevo fichero con el contenido de la salida, o bien añadir la salida a un fichero que ya existe. En este último caso utilizamos dos signos mayor que (>>) en vez de uno antes del nombre del fichero.
- Para **redireccionar la salida de errores** utilizamos los caracteres &> (o también >&) seguidos del nombre de fichero o dispositivo a donde irá a parar la información de los errores.

La tabla mostrada a continuación muestra algunos ejemplos de redirección.

A veces conviene que la salida de un programa se utilice como entrada de otro. Por ejemplo, nos puede interesar listar los archivos de un directorio, pero haciendo una pausa cada vez que se llene la pantalla². Podemos hacer la siguiente secuencia:

```
$ ls >temporal
$ more <temporal
```

Esta es una operación muy habitual; por eso existe un mecanismo que nos permite reunir los dos comandos en una única orden. En vez de utilizar un fichero auxiliar (`temporal` en el ejemplo) se utiliza lo que se denomina una **tubería** (*pipe*). Una tubería es como un fichero en el que un comando escribe y el otro lee, pero no tiene una representación real en el disco, sino que se crea en memoria de forma automática, y se destruye una vez terminada la orden. Para utilizar una tubería utilizamos el símbolo de raya vertical (`|`, *pipe*). El ejemplo anterior quedaría:

```
$ ls | more
```

Se pueden encadenar varios comandos con varias tuberías. Todos los comandos se ejecutan de forma concurrente, de forma que la salida de cada comando a la izquierda de un `|` se utiliza como entrada en el comando inmediatamente a la derecha de `|`.

² El comando `more` sirve para hacer que un fichero se muestra por páginas, haciendo una pausa cada vez que se llena una pantalla.

Comando	Significado
<code>ls >mifichero</code>	Lista los archivos del directorio y los escribe en el fichero <code>mifichero</code> . Si este fichero ya existe, el contenido anterior se pierde.
<code>ls >>mifichero</code>	Lista los archivos del directorio y los añade al fichero <code>mifichero</code> . Si este fichero no existe se crea.
<code>ls &>mifichero</code>	Lista los archivos del directorio en pantalla y escribe los posibles errores en el fichero <code>mifichero</code> . Si este fichero ya existe, el contenido anterior se pierde.
<code>ls &>/dev/null</code>	Lista los archivos del directorio en pantalla y descarta los posibles errores.
<code>more <mifichero</code>	Muestra en pantalla el contenido de <code>mifichero</code> haciendo una pausa en cada página.
<code>ls more</code>	Muestra en pantalla la lista de ficheros haciendo una pausa en cada página.

8.3.- PRECAUCIONES AL DIRECCIONAR LA SALIDA

Cuando se redirecciona una salida, el sistema UNIX crea un fichero con el nombre especificado. Cuando el fichero no existe, al redireccionar la salida a él crea uno nuevo; pero si el fichero existe borra su contenido y reescribe encima. Afortunadamente, existe una manera de prevenir borrar ficheros de esta forma sin darnos cuenta, utilizando el comando:

set noclobber

Si tecleamos este comando antes de redireccionar la salida, en el caso de que el fichero donde se envía la salida ya exista, aparecerá en la pantalla el siguiente mensaje:

nombre del fichero **file exists**

y no "machacaría" el contenido del fichero.

El comando **set noclobber** evita que se :

- redireccione una salida a un fichero ya existente
- añada un fichero a otro que no existe.

El comando sólo es efectivo para la sesión en que se teclee. Si se quiere que permanezca de forma permanente, hay que incluir el comando **set noclobber** en el fichero **.cshrc** del Home directory.

Si en algún momento se quisiera quitar esta protección, hay que teclear **>!**.

Veamos un ejemplo:

```
%set noclobber
%cat agenda
```



```

contestar carta al señor Alvarez
%date > agenda
agenda : file exists
%cat agenda
contestar carta al señor Alvarez
%date >! agenda
%cat agenda
Mon May 4 08:50:38 CDT 1989

```

8.4.- VARIABLES DE ENTORNO

Las variables de entorno son un conjunto de variables que están presentes en el sistema UNIX y que algunos programas, así como el propio shell, utilizan para intercambiar información, conocer datos de configuración, etc. Una variable se identifica mediante una cadena de caracteres, y su valor es siempre otra cadena de caracteres (recordemos una vez más que se diferencia entre mayúsculas y minúsculas).

Para definir el valor de una variable utilizamos una asignación (no necesitamos ningún comando). Por ejemplo, para definir la variable `TEMPORAL` con el valor `/tmp` hacemos:

```
| $ TEMPORAL=/tmp |
```

Para utilizar la variable, antepone el símbolo `$`. Por ejemplo, la siguiente orden muestra el contenido del directorio almacenado en la variable `TEMPORAL`:

```
| $ ls $TEMPORAL |
```

Al definir variables podemos utilizar otros caracteres especiales, como en cualquier comando. Por ejemplo:

```
| $ ESTE_DIR=`pwd`
| $ ls ${ESTE_DIR}/*.c |
```

En este ejemplo se han utilizado las llaves `{}` para delimitar el nombre de la variable. Si se hubieran omitido (`ls $ESTE_DIR/*.c`), el shell podría pensar que el nombre de la variable incluye, al menos, el barra de directorio (`/`), y entonces el valor de la misma no sería el esperado.

Si intentamos utilizar el valor de una variable no definida, obtendremos como resultado una cadena vacía (nunca un error).

Para utilizar el símbolo `$` en un comando sin que denote una variable, debemos encerrarlo entre comillas simples (`'`) o anteponer una barra inversa (`\`).

Las variables definidas en un shell son locales a ese shell. Para hacer que los programas lanzados por un shell conozcan el valor de esas variables es necesario *exportarlas*, con el comando `export` (ya lo estudiaremos).

Existen algunas variables predefinidas de uso muy común, que están reflejadas en la tabla que aportamos seguidamente. La más importante de todas ellas es la variable **PATH**, que almacena un conjunto de rutas de búsqueda para localizar comandos (*path de búsqueda*). Así, no es necesario escribir la ruta completa a un comando, siempre que esa ruta esté incluida entre las rutas de la variable **PATH**. **PATH** almacena un conjunto de rutas separadas por dos puntos (:).

Comando	Significado
PATH	Rutas de búsqueda de comandos.
HOME	Ruta al directorio <i>home</i> .
PS1	<i>Prompt</i> principal.
PS2	<i>Prompt</i> secundario.
SHELL	Comando utilizado como shell.
LOGNAME	Nombre utilizado en el <i>login</i> (nombre de usuario).
HOSTNAME	Nombre completo de la máquina UNIX

Variables de entorno más importantes

Podemos añadir nuevas rutas al path utilizando el valor de la variable. Por ejemplo, para añadir el directorio `/tmp` al path:

```
| $ PATH=${PATH}:/tmp |
```

Otras variables importantes son las que definen el *prompt*. Hasta ahora sabemos que el prompt sirve para que el shell nos indique que está listo para recibir comandos, y que suele ser un símbolo (p.e. `$`). En realidad existen varios prompts. El principal es el que conocemos. Pero a veces algunos comandos, al ejecutarlos piden más información por teclado, y entonces se muestra un prompt secundario, terciario, etc.

El aspecto de todos los prompts se almacena en las variables **PS1**, **PS2**, etc., siendo **PS1** el principal. En estas variables se almacena la cadena de caracteres que se muestra como prompt, aunque existen algunos códigos que tienen funciones especiales (como si fueran caracteres especiales). La tabla posterior recoge algunos de los códigos más habituales utilizados en las variables de prompt. Todos comienzan por el símbolo `\`, por lo que al definir la variable habrá que utilizar doble barra o encerrar el valor entre comillas. El siguiente ejemplo construye un prompt principal formado por el nombre de la máquina, seguido de dos puntos, el directorio de trabajo y un símbolo `$` o `#`:

```
| PS1="\h:\w\$ |
```

Código	Significado
\h	Nombre del computador UNIX.
\w	Ruta al directorio por defecto.
\t	La hora actual en formato HH:MM:SS.
\d	La fecha actual en formato " <i>DíaSemana Mes Día</i> " (por ejemplo "Tue May 26").
\u	El nombre del usuario.
\\$	El símbolo \$ si es un usuario normal o # si es el administrador.

8.5.- ERROR ESTÁNDAR

Hemos discutido anteriormente la entrada y salida estándar de las órdenes, pero el sistema también proporciona un tercer canal, el *error estándar*. La mayoría de las órdenes usan el canal de error estándar para mostrar los mensajes de error o la entrada inusual que no se desearía ver en el flujo de salida estándar. Nosotros no deseamos que los mensajes de error se mezclen con la salida normal, de modo que el sistema escribirá estos mensajes en otro flujo. Podemos ver el error estándar en operación si intencionadamente creamos una orden incorrecta. Por ejemplo, si tratamos de concatenar un fichero que no existe, veremos un mensaje de error:

```
$ cat no.fichero

cat: cannot open no.fichero

$
```

Si ahora redirigimos la salida estándar de la orden **cat**,

```
$ cat no.fichero > salida

cat: cannot open no.fichero

$
```

los resultados son los mismos. El mensaje de error continúa apareciendo en el terminal. Podemos redirigir el error estándar usando el operador **2>** o **2>>**, dependiendo de si queremos crear un fichero nuevo o añadir datos a un fichero existente, respectivamente.

```
$ cat no.fichero 2>salida

$ cat salida

cat: cannot open no.fichero

$
```

La peculiar notación **2>** se usa ya que los canales de E/S estándar son números asignados: 0 se refiere a la entrada estándar; 1 se refiere a la salida

estándar; y 2 se refiere al error estándar. Estos números son usados principalmente por los programadores de software.

8.6.- CONFIGURACIÓN DE OPCIONES DE *BASH*

El comportamiento del shell *bash* ante determinadas circunstancias puede ser modificado mediante la configuración de un conjunto de opciones mediante el comando `set`. Este comando lo estudiaremos más adelante como utilidad para listar las variables de entorno definidas, pero ahora lo vamos a utilizar como método de configuración.

bash proporciona un conjunto de opciones que pueden ser activadas o desactivadas. Por defecto aparecen desactivas, pero podemos activarlas mediante el comando `set`. Por ejemplo `set -x` activa la opción "`x`". Luego, con `set +x` podemos volver a desactivar la opción "`x`".

La opción `-x`, cuando es activada hace que *bash* muestre, antes de ejecutar un comando, la verdadera secuencia de parámetros que se ha pasado al comando. Esto incluye la sustitución de alias por el comando y opciones verdaderos, la sustitución de los nombres de variables por sus valores verdaderos, sustitución de máscaras y comodines por listas de ficheros, etc. De esta manera podemos analizar lo que realmente se ejecuta cuando invocamos un comando, lo cual nos puede dar una idea precisa de por qué ocurren algunas cosas "misteriosamente".

Por ejemplo, cuando invocamos `ls *.c`, realmente estamos haciendo mucho más que eso. Vemos, en la línea que comienza por `+` el verdadero comando, que incluye la ruta completa al comando, las opciones definidas mediante el alias y la sustitución de la máscara `*.c` por el conjunto de ficheros que casan.

```
Remigio:~$ ls *.c
+ /bin/ls --8bit --color=tty -F -b -T 0 busqueda.c cgi.c cgi00.c
libreria.c prueba.c
busqueda.c  cgi.c      cgi00.c     libreria.c  prueba.c
```

Igualmente, en el siguiente comando se puede ver la sustitución de una variable por su valor:

```
Remigio:~$ echo Mi directorio es $HOME
+ echo Mi directorio es /home/Remigio
Mi directorio es /home/Remigio
```

Otra opción interesante de *bash* es la `-v`. Es parecida a `-x`. Hace que antes de ejecutar un comando se muestre el comando que se va a ejecutar, pero a diferencia de `-x`, con `-v` no se realiza ninguna sustitución, sino que se muestra el comando tal cual se escribió. Esto no tiene ninguna utilidad mientras escribimos comandos, ya que volvemos a ver lo que acabamos de escribir. Pero resulta realmente interesante cuando ejecutamos *scripts* (programas hechos como secuencias de comandos), ya que entonces podemos hacer una traza de lo que se

está ejecutando en un script, que de otra manera no podemos ver. Estudiaremos los scripts más adelante.

La opción `-f` permite hacer que bash no realice ninguna sustitución de máscaras con comodines por nombres de ficheros que casen, sino que pasa a los comandos los comodines como caracteres normales. Por ejemplo:

```
Remigio:~$ set -f
Remigio:~$ ls *
/bin/ls: *: No such file or directory
```

Permisos sobre ficheros

9.1.- FUNDAMENTOS DE LOS DERECHOS DE ACCESO

Cada archivo del árbol de archivos de UNIX posee lo que se denomina datos de identificación de archivo. Los más importantes se pueden ver mediante el comando *ls*, acompañado de *-l*. Gracias a ello puede apreciarse que existen unos derechos de acceso para cada archivo, ya se trate de un archivo normal, un archivo de dispositivo o un directorio. Para cada archivo pueden establecerse unos derechos de acceso propios. Cada vez que se accede a un archivo, el sistema comprueba si el tipo de acceso que se intenta está permitido.

Sin embargo, los derechos de acceso que se guardan para cada archivo no bastan para garantizar su seguridad. Por ello, a cada usuario se le adjudica al identificarse un número único de usuario (UID). La asignación de nombres a los usuarios se lleva a cabo mediante el archivo **/etc/passwd**. Este archivo es un elemento básico en toda la gestión de usuarios. La forma de este archivo y sus distintos campos serán explicados en la sección de *Administración del sistema*. Además, en ese mismo archivo se asigna a cada usuario un número de grupo (GID). De este modo, cada usuario que accede al sistema posee un número de usuario y un número de grupo. Para saber qué números nos corresponden, podemos ejecutar el comando *id*.

A nivel interno, UNIX distingue a los diferentes usuarios sólo por los números, y no por sus nombres.

Después de la identificación, UNIX olvida el nombre del usuario y debe extraerlo cada vez que sea necesario del archivo **/etc/passwd**.

Lo mismo sucede con los archivos. A nivel interno, el sistema accede a ellos sólo a través de un número único. A través del número de archivo se accede a los datos de identificación del archivo, entre los que se cuentan

- el número de usuario del propietario del archivo y
- el número del grupo al que está asignado el archivo.

Con esto, cada vez que un usuario accede a un archivo, existen tres diferentes combinaciones posibles:

1.- El número del usuario coincide con el número de usuario almacenado en el encabezado del archivo. En tal caso, el usuario es el propietario del archivo.

2.- Si ambos números de usuario no son iguales, se comprueba si coinciden los números de grupo del usuario y del archivo. En caso afirmativo, el usuario pertenece al mismo grupo que el archivo.

3.- En todos los demás casos (es decir, si no coinciden ni el número de usuario ni el número de grupo), se habla de “resto del mundo”, o sea, el conjunto de usuarios que no son propietarios del archivo ni pertenecen al mismo grupo.

Estas tres categorías de usuario que se distinguen cada vez que se accede a un archivo son muy importantes.

9.2.- SIGNIFICADO DE LOS DERECHOS DE ACCESO

Cada fichero de UNIX (ficheros, directorios, dispositivos, etc.) tiene unos determinados *permisos*. Los permisos son de tres tipos, que pueden estar o no activados (si está activado, tenemos el permiso), y tienen un significado diferente para ficheros y directorios.

Para ficheros:

- **Permiso de lectura:** nos permite ver el contenido del fichero.
- **Permiso de escritura:** nos permite modificar el contenido del fichero.
- **Permiso de ejecución:** nos permite ejecutar el fichero (que se supone que es un programa).

Para directorios:

- **Permiso de lectura:** nos permite conocer el contenido del directorio (listado de ficheros), pero no permite ver información detallada (tamaño, fecha, etc.) de los ficheros a menos que tengamos también permiso de ejecución. Hay que distinguir entre poder saber qué ficheros hay (permiso de lectura sobre un directorio) y qué contiene cada fichero (permiso de lectura sobre cada fichero).
- **Permiso de escritura:** nos permite modificar el contenido del directorio: añadir ficheros y eliminar ficheros (siempre que tenga permiso de escritura sobre los ficheros). Recordemos que para poder modificar los contenidos de los ficheros se requiere permiso de escritura sobre los ficheros.
- **Permiso de ejecución:** nos permite que podamos cambiar el directorio por defecto a ese directorio, así como copiar ficheros del directorio (siempre que además tengamos los permisos de lectura de los ficheros).

Solemos referirnos a estos permisos de ficheros y directorios como *r*, *w* y *x* (lectura, escritura y ejecución, respectivamente). Así, si tenemos permisos *rx*, podemos *leer* y *ejecutar*, pero no *modificar*. Además, estos permisos se dan en función del usuario:

- **Permisos para el usuario propietario:** indican los permisos (*r*, *w*, *x*) que tiene el usuario propietario del archivo. El propietario de un archivo es el usuario que lo creó por primera vez, aunque es posible cambiar de propietario con el comando adecuado (`chown`).

- **Permisos para los usuarios del mismo grupo que el propietario:** indican los permisos (*r*, *w*, *x*) que tienen los usuarios que pertenecen al mismo grupo que el usuario propietario del archivo.
- **Permisos para los otros usuarios:** indican los permisos (*r*, *w*, *x*) que tienen los usuarios que no pertenecen al grupo del usuario propietario.

Solemos referirnos a estos permisos como *U*, *G* y *O* (usuario –propietario–, grupo y otros, respectivamente). También se suelen representar con variantes de la cadena *rwxtwxrwx*, donde los permisos aparecen en el orden *UGO*, de manera que si el permiso existe se pone la letra correspondiente, y si el permiso no existe se pone un guión. Por ejemplo: *r--r--r--*, *rwxt--xt--x*, *rwxt-----*, etc.

Los permisos de un fichero pueden cambiarse, con el comando `chmod`, por usuario propietario y por el administrador, aunque no tengan permiso de escritura sobre el fichero.

Habitualmente, junto con los permisos se suele representar el tipo de fichero utilizando un carácter que se coloca antes de los permisos: directorio (*d*), fichero (*-*), dispositivo de caracteres (*c*), dispositivo de bloques (*b*), enlace simbólico (*l*), etc. Por ejemplo, *drwxr-xr-x* son los permisos de un directorio, mientras que *-rwxr-xr-x* son los permisos de un fichero normal.

9.3.- MODIFICACIÓN DE LOS DERECHOS DE ACCESO

El comando `chmod` (*change mode*) sirve para cambiar los permisos de un fichero o directorio. Hemos visto los permisos en la sección anterior. `chmod` admite como parámetro, primero los nuevos permisos, y luego los archivos (un fichero, una máscara o una lista) a los que hay que aplicar los permisos:

Sintaxis

chmod [*opciones*] permisos ficheros

Los permisos se pueden especificar de dos formas: simbólica o numéricamente. En **notación simbólica**, los permisos se especifican con la siguiente sintaxis:

$[\{u|g|o|a\}]\{+|-|= \}\{r|w|x|s\}$

En concreto: cero o más letras del conjunto $\{u, g, o, a\}$, un símbolo de $\{+, -, =\}$, y una letra de $\{r, w, x, s\}$.

- La primera letra identifica los permisos a cambiar: propietario (*u*), grupo (*g*), otros usuarios (*o*), o bien todos los permisos (*a*). Si se omite esta parte, se supone *a*.
- El símbolo identifica cómo se cambiarán los permisos: añadir un permiso (+), restringir un permiso (-), o bien establecer un permiso único (=).
- La última letra identifica el permiso asignado: de lectura (*r*), escritura (*w*), de ejecución (*x*), o de establecimiento de propietario (*s*). Si se trata de este último (*s*), la primera letra sólo puede identificar al usuario (*u*) o al grupo (*g*).

Por ejemplo:

- Añadir permiso de lectura a todos los usuarios: `a+r`.
- Restringir permiso de escritura a los usuarios que no son del grupo: `o-w`.
- Establecer permisos `rx` (lectura y ejecución) al propietario: `u=rx`.
- Añadir permisos de escritura al propietario y el grupo: `ug+w`.
- Añadir permisos de modificación de propietario y grupo: `ug+s`.

Podemos incluir varias claves a la vez separándolas mediante comas ***sin dejar espacios entre cada clave***. Por ejemplo, para establecer permisos de lectura y ejecución al propietario y ejecución a todos utilizamos la siguiente clave: `a=x,u+r`.

En ***notación numérica***, los permisos se especifican mediante un número octal de hasta 4 cifras, donde cada cifra (número entre 0 y 7) representa los permisos de establecimiento de propietario, propietario, grupo y otros, respectivamente, de forma el valor 0-7 indica los permisos según la clave de la tabla que sigue. En el caso del establecimiento de propietario, las claves son distintas.

Número	Valor binario	Permiso
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwX

Si se especifican menos de 4 cifras, se supone que las primeras son 0, es decir que 174 es equivalente a 0174. Los permisos de establecimiento de propietario no suelen utilizarse, por lo que normalmente se emplean sólo 3 dígitos, para los permisos de acceso a la información, que son los más comúnmente utilizados. Veamos algunos ejemplos:

- Otorgar todos los permisos: 777.
- Permiso de lectura a todos, y de lectura, ejecución y escritura al propietario: 744.
- Permiso de lectura y ejecución a todos: 555.
- Permiso de lectura y escritura al usuario, de ejecución y lectura al grupo y de ejecución al resto: 651.

Los modificadores que admite el comando `chmod` se resumen en la tabla que se expone:

Modificador	Resultado
-v	Muestra el cambio que realiza en los permisos cada fichero procesado.
-c	Muestra el cambio que realiza en los permisos cada fichero procesado sólo si realiza algún cambio.
-f	No muestra errores cuando no sea posible cambiar los permisos.
-R	Recurso: procesa también los ficheros que estén dentro de subdirectorios.

9.4.- MODIFICAR LOS DERECHOS DE ACCESO PREESTABLECIDOS

Después de que hayamos creado un fichero, deberíamos verificar los permisos con **ls -l** para asegurarnos que los permisos sean los que deseamos. Debemos preguntarnos si ese fichero cumple nuestras necesidades o, si por el contrario, debemos restringir o aumentar el tipo de acceso.

El sistema UNIX crea un fichero propiedad del creador del fichero (y del grupo del creador), y esto no puede ser alterado. Sin embargo, podemos preparar una variable del sistema asociada con nuestro *id* de presentación que establece los permisos de un fichero sin acción explícita por nuestra parte. Esta variable del sistema se denomina **umask**, y es concedida con la orden **umask**. Podemos determinar el valor actual de **umask** ejecutando la orden sin argumentos:

```
$ umask
000
$
```

El resultado son tres dígitos octales que se refieren a los permisos del propietario, el grupo y los otros, el resto (de izquierda a derecha). A este número se le denomina *máscara*, ya que cada dígito se resta de un permiso implícito global del sistema que todos los ficheros nuevos obtienen. Normalmente este permiso global es **-rw-rw-rw-**, pero sistemas y programas individuales pueden diferir de este valor. Puesto que la **umask** del usuario se resta de este valor implícito, no podemos desactivar permisos con **umask** que estén normalmente desactivados. Pero podemos desactivar permisos que están normalmente activados. Naturalmente, podemos activar explícitamente permisos con **chmod** (vista en el apartado anterior), si tenemos la propiedad del fichero.

Cada dígito octal en la **umask** contiene un bit binario que borra un permiso: un 1 borrará el permiso de ejecución; un 2 borrará el permiso de escritura; y un 4 borrará el permiso de lectura. Por tanto, si un dígito es 0, entonces se usa el permiso implícito. Por ejemplo, la **umask** cada anteriormente (000) significa no alterar ninguno de los valores implícitos. Si el valor **umask** fuera 022, entonces los ficheros se crearían sin permiso de escritura para el grupo o para otros. Por ejemplo,

```
$ umask
000
```

```

$ >def.perm

$ ls -l def.perm

-rw-rw-rw- 1 Remigio      users  0 May 10 14:57 def.perm

$ umask 022

$ umask

022

$ >no.escrib

$ ls -l no.escrib

-rw-r--r--  1 Remigio      users  0 May 10 14:58 no.escrib

$ umask 777

$ >no.perm

$ ls -l no.perm

----- 1 Remigio      users 0 May 10 14:58 no.perm

$

```

Con la **umask** implícita de 000, los ficheros se crean con los permisos implícitos. Cuando redefinimos nuestra **umask** a 022, creamos ficheros sin permiso reescritura para el resto de usuarios. Una especificación de 777 desactiva todos los permisos para todos los usuarios, por lo que el último fichero no es accesible en absoluto.

Normalmente la **umask** está definida en nuestro **.profile**, o quizás en el **/etc/profile** global del sistema, dependiendo de la política de seguridad de cada sistema. De este modo, no necesitamos considerar los permisos para cada fichero que creamos.

9.5.- DEPENDENCIAS

Es importante darse cuenta de que los permisos de un fichero también dependen de los permisos del directorio en el que residen. Por ejemplo, aunque un fichero tenga los permisos **-rwxrwxrwx**, otros usuarios no podrán acceder a él a menos que también tengan permiso de lectura y ejecución para el directorio en el cual se encuentra el fichero. Si Remigio quiere restringir el acceso a todos sus ficheros, podría simplemente poner los permisos de su directorio “home” **/home/Remigio** a **-rwx-----**. De esta forma ningún usuario podrá acceder a su directorio ni a ninguno de sus ficheros o subdirectorios. Remigio no necesita preocuparse de los permisos individuales de cada uno de sus ficheros.

En otras palabras, para acceder a un fichero, debemos tener permiso de ejecución de todos los directorios a lo largo del camino de acceso al fichero, además de permiso de lectura (o ejecución) del fichero en particular.

Habitualmente, los usuarios de un sistema UNIX son muy abiertos con sus ficheros. Los permisos que se dan a los ficheros usualmente son **-rw-r--r--**, lo que permite a todos los demás usuarios leer los ficheros, pero no modificarlos de ninguna forma. Los directorios usualmente tienen los permisos **-rwxr-xr-x**, lo que permite que los demás usuarios puedan moverse y ver los directorios, pero si crear o borrar nuevos ficheros en ellos.

Muchos usuarios pueden querer limitar el acceso de otros usuarios a sus ficheros. Poniendo los permisos de un fichero a **-rw-----** no se permitirá a ningún otro usuario acceder al fichero. Igualmente, poniendo los permisos del directorio a **-rwx-----** no se permitirá a los demás usuarios acceder al directorio en cuestión.

Montajes de dispositivos y unidades de ficheros

El usuario que se acerque por primera vez a UNIX se sorprenderá bastante cuando encuentre que no puede usar un disquete libremente, como hace en MS-DOS o Windows, sino que antes debe montarlo sobre el sistema de ficheros existente en UNIX. Al igual sucede con el CD-ROM, por ejemplo. Esta característica es única de UNIX (y también, por supuesto de UNIX), y veremos porqué

10.1.- INTRODUCCIÓN

Ya sabemos que el UNIX todas las unidades de disco son accedidas como si formaran una única unidad, organizada en un único árbol de directorios. Como es posible que se añadan nuevas unidades una vez arrancado el sistema (por ejemplo, si introducimos un disquete o un CD-ROM), es necesario indicar a UNIX en qué parte del árbol de directorios queremos situar el árbol contenido en la unidad de disco incorporada (montar la unidad), y también hay que advertir al sistema de que vamos a retirar el contenido de la unidad (desmontar la unidad). Si no hacemos correctamente estas operaciones (por ejemplo, retiramos un disquete sin desmontarlo previamente), podemos dañar irreparablemente el árbol de directorios y perder la información. Por esta razón, sólo se permite al administrador trabajar con las unidades de disco.

Antes de comenzar a estudiar los comandos debemos conocer cómo se denominan las distintas unidades de disco en UNIX. Esta denominación puede cambiar de unas versiones de UNIX a otras. La que sigue recoge la denominación en los sistemas UNIX convencionales.

NOTA:

Los nombres de las unidades son los dispositivos, unos ficheros especiales que se ubican en el directorio `/dev`. Los dispositivos son de dos tipos (de bloques, como un disco; y de caracteres, como un terminal). Las entradas de `/dev` asocian a cada nombre de dispositivo dos números que sirven para el sistema localice el hardware correspondiente. Cuando nos refiramos a un dispositivo, nosotros especificaremos la entrada adecuada del directorio `/dev`.

Dispositivo	Resultado
hda	Primer disco duro (IDE).
hda1	Primera partición del primer disco duro (IDE).
hda2	Segunda partición del primer disco duro (IDE).
hda3	Tercera partición del primer disco duro (IDE).
...	
hdb	Segundo disco duro (IDE).
hdb1	Primera partición del segundo disco duro (IDE).
...	
hdc	Tercer disco duro (IDE).
hdd	Cuarto disco duro (IDE).
...	
fd0	Primera unidad de disco flexible.
fd1	Segunda unidad de disco flexible.
...	

Así, de la misma manera que podemos montar una unidad física, también podemos hacerlo con una partición de nuestro disco duro. Por ejemplo, supongamos que tenemos Windows en alguna partición. Ejecutando la orden *mount* (que veremos a continuación), podemos montar ese sistema de ficheros en un punto del directorio de UNIX y acceder a él en cualquier momento. Es importante desmontar todos los puntos de montaje antes de finalizar cada sesión.

10.2.- EL MONTAJE. *MOUNT*

El comando *mount* nos permite montar una unidad al sistema de ficheros. Debemos indicar qué unidad montamos y en qué directorio del árbol queremos situar el sub-árbol contenido en dicha unidad.

Sintaxis

mount [*opciones*] [*dispositivo directorio*]

- *dispositivo* es alguna de las entradas de */dev* que identifican algún dispositivo de disco, por ejemplo los recogidos en la **¡Error! No se encuentra el origen de la referencia.**
- *directorio* es algún directorio que no esté en uso. No tiene necesariamente que estar vacío, pero mientras se haga uso del dispositivo (mientras siga montado), no podremos acceder al contenido anterior de ese directorio; una vez desmontado el dispositivo, el directorio quedará como estaba.

Si no especificamos dispositivo ni directorio, `mount` imprimirá un listado de los dispositivos actualmente montados.

UNIX soporta varios sistemas de ficheros (formatos de discos de distintos sistemas operativos), por lo que es posible que también tengamos que indicar el tipo de formato que tiene el contenido de la unidad montada. Esto lo hacemos mediante la opción `-t` seguida (con una separación) de una palabra clave que describe el tipo de sistema de ficheros: `minix`, `ext`, `ext2`, `xiafs`, `hpfs`, `msdos`, `umdos`, `vfat`, `proc`, `nfs`, `iso9660`, `smbfs`, `ncpfs`, `affs`, `ufs`, `romfs`, `sysv`, `xenix`, `coherent`. Normalmente UNIX detecta automáticamente el tipo de sistema de ficheros, pero a veces es necesario indicarlo explícitamente.

Normalmente los dispositivos sólo pueden ser montados por el administrador, aunque todos los usuarios pueden acceder al contenido de los dispositivos (siempre que tengan los permisos adecuados sobre los directorios de montaje). En este caso, una opción interesante es la de montar los dispositivos como sólo lectura con la opción `-r`, de forma que su contenido no se puede modificar.

El siguiente ejemplo monta la primera partición del primer disco duro en el directorio `/mnt`:

```
| mount /dev/hda1 /mnt |
```

En el siguiente caso, montamos un disquete del sistema operativo Minix en `/mnt`:

```
| mount -t minix /dev/fd0 /mnt |
```

La tabla siguiente contiene las opciones estudiadas para el comando `mount`.

Modificador	Resultado
<code>-r</code>	Monta el dispositivo como sólo lectura.
<code>-t</code>	Permite especificar a continuación el tipo de sistema de ficheros contenido en la unidad.

10.3.- EL DESMONTAJE. *UMOUNT*

El comando `umount` sirve para desmontar un dispositivo previamente montado con `mount`. Debemos especificar el dispositivo de que se trata, o bien el directorio dónde se montó.

Sintaxis

umount [*opciones*] [*dispositivo / directorio*]

Un dispositivo no puede ser desmontado cuando está **ocupado** (*busy*), es decir, cuando contiene ficheros abiertos, o cuando existan procesos cuyo directorio por defecto pertenezca al dispositivo.

Los siguientes ejemplos desmontan los ejemplos de montaje vistos con el comando `mount`:

```
umount /dev/hda1
umount /dev/fd0
```

10.4.- EL COMANDO *DF*

El comando `df` (*disk free*) sirve para visualizar el estado de los discos que se encuentran montados en el sistema: tipo, capacidad.

Sintaxis

df [*opciones*]

Por ejemplo, el siguiente listado indica que el sistema de ficheros de Remigio está formado por dos discos (`sda2`, montado como raíz, y `sda3`, montado como `/home`). El primer disco tiene una capacidad de 1.982.006 bloques de 1024 bytes³ (es decir, 1,89 GB), del que se han ocupado 684.010 bloques (el 36%) y quedan libres 1.195.545 bloques. El segundo disco tiene una capacidad de 2.191.875 bloques (2,09 GB), del que se han ocupado 1.218.246 (el 59%) y quedan libres 860.324.

```
Remigio:~$ df
Filesystem            1024-blocks  Used Available Capacity Mounted on
/dev/sda2              1982006   684010   1195545     36%    /
/dev/sda3              2191875  1218246    860324     59%   /home
```

Todo esto quiere decir que disponemos de 840 MB libres para escribir archivos dentro del directorio `/home` (y sus subdirectorios) más 1,14 GB para escribir en el resto de directorios del disco.

10.5.- EL COMANDO *MDIR*

El comando `mdir` (*MS-DOS directory*) sirve para visualizar el contenido de un disquete (o en general de cualquier disco) de MS-DOS/Windows sin necesidad de montar ni desmontar la unidad de disquetes (la unidad no debe estar montada

³ A veces se utilizan bloques de 512 bytes a menos que se indique lo contrario mediante alguna opción

para que funcione). El listado se muestra con el mismo aspecto a como lo hace el comando `dir` de MS-DOS/Windows.

Sintaxis

mdir [*opciones*] directorio

El directorio se especifica mediante una mezcla de formato entre UNIX y MS-DOS/Windows:

- El nombre de la unidad se especifica mediante la letra de MS-DOS/Windows, seguida de dos puntos (:).
- La barra de separación de directorios no es inversa (\), sino que es la que se emplea en UNIX (/).

El siguiente ejemplo visualiza el contenido del directorio copia que se encuentra en el disquete MS-DOS introducido en la primera unidad de disquetes:

```
mdir a:/copia/*.*
```

Los modificadores que acepta el comando se resumen en la tabla siguiente:

Modificador	Resultado
-w	Muestra el listado en formato ancho (como /w en MS-DOS/Windows)
-a	Incluye los ficheros ocultos.
-f	No muestra información sobre el espacio libre en el disco (así es más rápido).

10.6.- EL COMANDO *MCOPY*

El comando `mcopy` (*MS-DOS copy*) sirve para copiar ficheros entre el árbol de directorios de UNIX y un disquete (disco en general) con formato MS-DOS/Windows sin necesidad de montar ni desmontar la unidad de disquetes (la unidad no debe estar montada para que funcione). La copia se puede realizar desde o hacia un disco MS-DOS/Windows.

Sintaxis

mcopy [*opciones*] origen [*destino*]

La composición de nombres de ficheros/directorios para las rutas MS-DOS/Windows sigue las mismas reglas que para el comando `mdir`.

Podemos especificar varios ficheros de origen, y en este caso, el destino será un directorio. También podemos especificar un fichero como origen y un fichero como destino, y entonces se copiará con el nuevo nombre. Finalmente podemos especificar sólo un fichero de origen, y el destino será el directorio por defecto.

Las opciones que admite `mcopy` son las resumidas en la tabla que se adjunta.

Modificador	Resultado
-t	Copia ficheros de texto (hace conversión de CR a CR+LF, y viceversa).
-n	No pide confirmación antes de sobrescribir ficheros UNIX.
-o	No pide confirmación antes de sobrescribir ficheros MS-DOS/Windows.

10.7.- EL COMANDO *mkfs*

El comando `mkfs` (*make file system*) sirve para crear discos con formato UNIX, aunque en UNIX se pueden especificar otros sistemas de ficheros.

<i>Sintaxis</i>
mkfs [<i>opciones</i>] dispositivo

Dispositivo identifica alguno de los dispositivos de disco (con posibilidad de escritura). Para seleccionar un dispositivo, recomendamos volver a la tabla realizada con ocasión de la exposición del montaje de archivos, al comienzo de este capítulo.

Con el parámetro `-t` podemos indicar a continuación el tipo de sistema de ficheros (como en el caso de `mount`). Además, con la opción `-c` hacemos que se verifique la superficie del disco en busca de bloques dañados.

Modificador	Resultado
-t	Permite especificar a continuación el tipo de sistema de ficheros contenido en la unidad.
-c	Comprueba los bloques defectuosos.

Impresión

*El sistema UNIX proporciona buenas herramientas para controlar impresoras y hacer salida por spool a dispositivos de copia impresa. Muchos usuarios pueden dirigir salida a una impresora simultáneamente; el software pondrá en cola la salida correctamente y añadirá páginas insignia a la impresión de modo que los usuarios individuales puedan encontrar su propia salida. Las herramientas de impresión se denominan genéricamente subsistema **lp**. Son tan generales y tan potentes que en los sistemas UNIX no se dispone generalmente de ningún otro software de impresión.*

11.1.- USO DE LA ORDEN **LPR**

El acceso primario a nivel de usuario al sistema **lp** se hace con la orden **lp**. Frecuentemente se usará **lp** como última orden en una línea de cauce shell,

```
$ cat /etc/passwd | lp
```

pero también puede tomar un nombre de camino como argumento.

```
$ lp /etc/passwd
```

La orden **lp** es especialista en imprimir; si deseamos paginar la salida con cabeceras especiales en cada página, usaremos una herramienta adicional apropiada en la línea de orden.

```
$ pr /usr/spool/lp/model/serial | lp
```

Por omisión, **lp** colocará el fichero o su entrada estándar en la cola de impresión para la impresora solicitada. La orden **lp** regresará al shell después de que el trabajo haya sido colocado en la cola, no cuando se haya completado la impresión.

lp

Sintaxis

lp [opciones] [archivo]

La orden **lpr** admite las siguientes opciones, según indica la tabla siguiente:

Modificador	Resultado
-m	Facilitará notificación por correo electrónico después que el fichero haya sido impreso.
-n	Permite imprimir más de una copia
-t	Controla los contenidos de la página insignia de cada trabajo de salida
-P	Sin introducir esta opción, se imprime en la impresora predeterminada. Si se desea imprimir en otra impresora, deberá introducirse el nombre de la impresora deseada tras la opción <i>-P</i>
-d	Toma el nombre de la impresora especificada como argumento
ficheros	Nombre o nombres de los ficheros que se van a imprimir
-w	Escribe un mensaje directamente en nuestro terminal cuando concluya la impresión
-c	Copia ficheros antes de imprimirlos
-o	Transmite opciones específicas de la impresora directamente al programa real que controla la impresora.

Por ejemplo:

```
lp -#3 -p datos.dat
```

imprime 3 copias del fichero *datos.dat* y además, en el principio de cada página imprime la fecha, el nombre *datos.dat* y el número de página.

11.2.- EL *ID* DE PETICIÓN

Cuando la orden **lp** pone en cola un trabajo de impresión, devuelve un *id* de petición a su salida estándar.

```
$ lp /usr/spool/lp/model/prx
```

```
Request id is ATT470-78
```

```
$
```

Este número deberemos anotarlo porque puede servir para “seguir la pista” al trabajo o para cancelarlo. Algunas versiones de **lp** en algunas máquinas no devuelven el *id* de petición, pero aún así podemos seguir la pista a nuestro trabajo con la orden **lpstat**, que veremos a continuación.

Si disponemos de varias impresoras conectadas a nuestra máquina, podemos dirigir **lp** al uso de una impresora predeterminada usando la opción *-P*.

11.2.- CANCELACIÓN DE UN TRABAJO

Podemos cancelar un trabajo de impresión antes de que haya sido impreso con la orden **cancel**. La orden **cancel** toma un id de petición de impresora, o una lista de ids de petición, y suprime los trabajos nombrados de la cola de impresión.

```
$ cancel ATT470-78

request "ATT470-78" canceled

$
```

Esto cancelará un trabajo incluso si ha comenzado a imprimirse, permitiéndonos detener trabajos largos pero erróneos sin obstruir la impresora.

La orden **cancel** puede tomar también un nombre de impresora como argumento, en cuyo caso cancelará el trabajo que actualmente esté imprimiéndose sobre el dispositivo. Sin embargo, esto no afectará a los trabajos de la cola que no hayan comenzado a imprimirse.

Esta orden admite una serie de parámetros:

<i>Sintaxis</i>
cancel <i>identificador</i>

Como opciones posibles se permite sólo la introducción de *-p* con el nombre de la impresora.

Si tras *-P* y la impresora se especifica sólo un signo menos, se suprimirán todas las tareas de impresión de la cola de espera indicada. De lo contrario, se suprimirán las tareas de impresión que tienen el número de tarea o que pertenecen al usuario introducido como parámetro.

El *identificador* puede ser un nombre de usuario, un nombre de fichero o un número de trabajo, obtenido con la orden **lpstat**, explicada a continuación.

11.3.- EL STATUS DE LA IMPRESORA

La orden **lpstat** proporciona información sobre el estado general de nuestro sistema **lp**. Si la ejecutamos sin argumentos, **lpstat** facilita información acerca de nuestros trabajos en cola de impresión.

```
$ lp /etc/profile

Request id is ATT470-79

$ lpstat

ATT470-79      lp      4290    Apr 27 19:07

$
```

El id de petición es lo primero, seguido por el dispositivo hardware que procesará la petición, el tamaño de la salida en bytes y, finalmente, la fecha y hora de la petición. Podemos usar **lpstat** de este modo para determinar los ids de petición de nuestros trabajos si llegáramos a olvidarlos.

Análogamente, podemos usar la opción **-u** con un id de usuario para ver las peticiones en spool de otros usuarios.

Cuando un trabajo está imprimiéndose, esa información está incluida en la salida de **lpstat**.

Cuando el trabajo ha acabado de imprimirse, se suprime de la cola y ya no habrá ningún modo de “seguirle la pista”. Sin embargo, si sabemos que el trabajo estaba en spool, su desaparición del spool indica que el trabajo ya ha sido impreso.

La orden **lpstat** admite los siguientes parámetros:

Modificador	Resultado
-d	Da cuenta de la impresora implícita
-r	Nos dice si el sistema de impresora está en operación. Si el sistema lp no está disponible, se informará de este hecho.
-t	Nos da completa información acerca del sistema de impresora.

11.4.- EL DEMONIO DE LP: *LPSCHED*

El sistema **lp** está controlado por un proceso demonio denominado **lpsched**, que se ejecuta todo el tiempo que el sistema **lp** está activo. La orden **lpsched** maneja también la gestión de cola, para impedir que múltiples trabajos creados al mismo tiempo compitan por los recursos de la impresora, y gobierna los dispositivos de impresión, detectando cuándo una impresora está ociosa o no funciona.

Podemos ver el programa **lpsched** con la orden **ps -ef**. Aparecerá una lista en pantalla mostrando todos los procesos que se encuentran activos en el sistema, junto con su UID y su PID. El padre del proceso **lpsched** es **init**, y no tiene **tty**. Pertenece al usuario **lp**. Generalmente no debemos intentar presentarnos ante el sistema como **lp**, y es deseable desactivar la contraseña para la identificación **lp** en **/etc/passwd** por razones de seguridad.

11.5.- CONFIGURACIÓN DEL SOFTWARE *LP*

Se usa la orden **/usr/lib/lpadmin** para preparar y modificar la configuración de las impresoras. Con esta orden podemos añadir impresoras, definir el tipo de impresora para el sistema **lp**, asignar impresoras a clases y capacitar o descapacitar lógicamente las impresoras. El uso de esta orden está reservada, como es lógico, y por motivos de seguridad, al **root** o superusuario.

Debemos asegurarnos de que el programa **lpsched** no esté corriendo cuando usemos **lpadmin**. Primero desactivaremos **lpadmin** con la siguiente orden:

```
# /usr/lib/lpshut
```

Luego podremos administrar nuestras impresoras.

El programa **lpadmin** es un programa sofisticado que puede aceptar muchas opciones.

Sintaxis

lpadmin [*opciones*] impresora destino

La función principal de **lpadmin** es tomar una impresora *destino* y emparejar ese nombre con un modelo y un fichero de dispositivo. Podemos asignar una impresora con el nombre que queramos, pero ya que se trata del nombre público usado con nuestras órdenes, deberíamos darle un nombre que tenga cierta relación con el tipo de impresora.

Modificador	Resultado
-p	Se usa para referirnos a una impresora en concreto.
-m	Toma el nombre del guión modelo al instalar una nueva impresora.
-v	Toma el nombre de camino completo del fichero de dispositivo que deseamos usar.
-d	Hace que nuestra impresora sea considerada como implícita, por defecto.
-x	Elimina una impresora del sistema.

11.6.- ACEPTACIONES DE PETICIONES DE IMPRESIÓN

Cuando añadimos una nueva impresora, queda instalada pero no activada. Hay dos características más del sistema **lp** de interés aquí. Podemos configurar una impresora para que “acepte” o “rechace” peticiones de impresión sin retirarla del sistema. El resultado visible es que la orden **lp** rehusará colocar un trabajo en spool cuando el dispositivo no esté aceptando peticiones de impresión. Se usa la orden **/usr/lib/accept/** para permitir que un destino de impresora acepte peticiones.

```
# /usr/lib accept MiIMPR

destination "MiIMPR" now accepting requests

# lpstat -t

scheduler is not running

system default destination: ATT470
```

```

members of class Parallel:

    ATT470

Device for ATT470: /dev/lp

Device for MiIMPR: /dev/tty02ATT470 accepting requests since Aug 19
18:58

Parallel accepting requests since Aug 19 18:58

MiIMPR accepting requests since May 1 18:33

Printer ATT470 is idle. Enabled since Apr 30 17:56

Printf MiIMPR disabled since Apr 30 19:00

    New printer

#

```

MiIMPR no aceptará peticiones para impresión.

Podemos usar **/usr/lib/reject** para decir a una impresora que rechace peticiones. Normalmente se usa esta orden cuando una impresora está fuera de servicio durante un tiempo relativamente largo, pero no se desea eliminarla del sistema. Al orden **/usr/lib/reject** toma un destino de impresora como argumento y, opcionalmente, una razón para desactivarla, a continuación de **-r**.

La razón será mostrada en la salida de **lpstat -t** y aparecerá también si un usuario trata de imprimir en el dispositivo con la orden **lp**.

11.7.- CAPACITACIÓN DE UNA IMPRESORA

Una impresora que esté aceptando peticiones puede no ser aún completamente funcional. Los trabajos se colocan en el spool para esa impresora. Podemos ver por la salida de **lpstat -t** que la impresora está todavía discapacitada (*disabled*). Debemos “capacitarla” (*enabled*) antes de que pueda imprimir realmente. A diferencia de la condición de aceptación/rechazo, el software **lp** discapacitará automáticamente una impresora si trata de imprimir en ella sin éxito. Si una impresora se queda sin papel o si la impresora está desenchufada, el dispositivo será discapacitado automáticamente. Podemos capacitar la impresora con la orden **enable**.

Generalmente, **disable** se usa para detener temporalmente la salida a un impresora mientras se añade papel o se desconecta la potencia.

11.8.- LA ORDEN *PR*

Aunque no es una orden estrechamente relacionada con la impresión, **pr** se usa bastantes veces junto a **lp**. Así que hemos decidido introducirla aquí y abordarla en el mismo apartado que la impresión.

pr [*opciones*] [*ficheros*]

El comando pr envía el contenido del fichero nombrado a la salida estándar (por defecto la pantalla). Si no hay opciones, el texto se organizará en páginas de 66 líneas, cada página comienza con una cabecera de 5 líneas y termina con otras cinco líneas de pie de página.

Estas últimas constan de líneas en blanco, la cabecera consta de dos líneas en blanco, una línea con el número de página, la fecha y hora y el nombre de fichero. Después vienen dos líneas más en blanco, las líneas que sean demasiado largas se dividirán en dos, la entrada estándar se utilizará cuando no se especifique fichero o cuando se usa un guión como nombre de fichero.

Algunas opciones producen columnas separadas. Por defecto, estas columnas tendrán el mismo ancho e irán separadas al menos por un espacio.

Modificador	Resultado
-p	Cuando la salida está dirigida a un terminal, se detiene antes del comienzo de cada página. El comando pr hará que el terminal genere un pitido y esperará hasta que usted teclee <INTRO> antes de continuar.
-lk	Especifica la longitud de página en k líneas. El valor por defecto es 66. Si no quedan suficientes líneas para incluir la cabecera y pie de página, estos serán suprimidos.
-h "cabecera"	Sustituye el nombre de fichero de la cabecera por la cadena <i>cabecera</i> .
-d	Doble espacio en la salida.
-t	Suprime las cinco líneas de cabecera y las cinco de pie de página. Termina la presentación después de la última línea de un fichero en vez de rellenar el final de la página con líneas vacías.
-wk	Especifica el ancho de línea a k posiciones de carácter para salida multicolumna. El valor por defecto es 72.
-nck	Provoca una salida no numerada. Si el entero k está presente, se emplearán k dígitos para cada número. El valor por defecto es 5. Si el carácter no numérico c está presente, se utilizará para separar los números del texto siguiente. El valor por defecto es el carácter fabulador. Las primeras k+1 posiciones de carácter de cada columna de salida se usarán para el número, y para la salida -m la línea se numerará por completo.

Procesos en UNIX

Un proceso o tarea es una instancia de un programa en ejecución. El shell de presentación es un proceso mientras estamos trabajando ya que siempre está presente hasta que nos despedimos. Si ejecutamos una orden ante “\$”, esa orden es un proceso mientras se está ejecutando. Los procesos tienen muchas propiedades y hay muchas órdenes para manipular los procesos y sus propiedades

12.1.- INTRODUCCIÓN

Cada vez que se comienza un trabajo, se dice que ha comenzado un proceso, que morirá en el momento en que el trabajo termine.

Se puede definir proceso como una unidad de ejecución desencadenada como consecuencia de la ejecución de una orden, un programa, o un módulo de programa.

Por tanto podemos decir que un proceso es una entidad activa y un programa es una entidad pasiva.

Gracias a los procesos UNIX puede trabajar en multiprogramación. Por ejemplo la orden: **cat /etc/passwd**, genera un proceso que subsiste, hasta que la orden se completa. Sin embargo, la orden: **cat /etc/passwd | more** genera dos procesos o unidades de ejecución, uno por cada orden.

El procesamiento en UNIX tiene lugar en *tiempo compartido* (distribución de la asignación de procesador entre los diferentes procesos que compiten por ejecutarse). Generalmente, los sistemas UNIX solo tienen una CPU que ejecuta los programas, y por tanto, solo un programa podrá estar ejecutándose en un momento determinado.

Una función muy importante del núcleo es proporcionar control y soporte a los muchos programas (los del sistema y los proporcionados por los usuarios) que pudieran desear utilizar la CPU en un momento dado. Es decir, existe una distribución temporal de la asignación de CPU entre los diferentes procesos que compiten por ejecutarse.

Así, solo se puede considerar un proceso cuando la orden que lo genera está en ejecución real o en espera de acceso a la CPU.

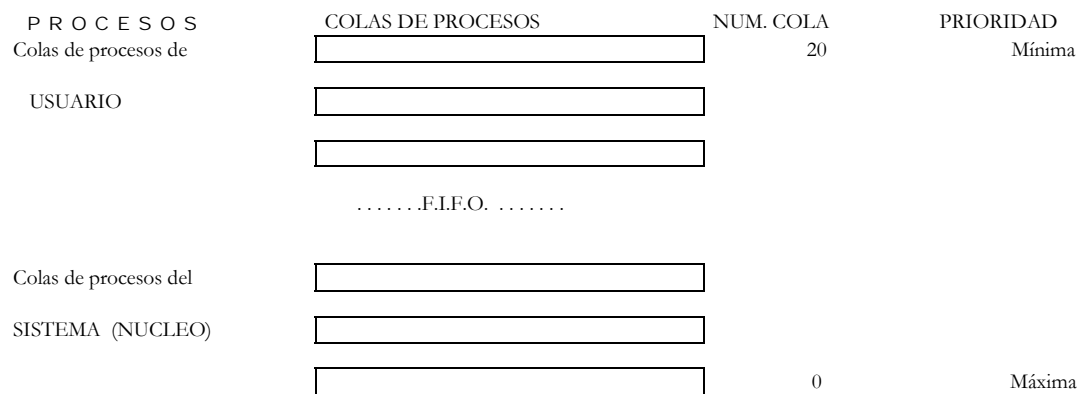
12.2.- CONTROL DE PRIORIDAD DE PROCESOS

Supuesto un ordenador con un solo procesador, el sistema UNIX realiza una distribución temporal de la asignación de CPU entre los diferentes procesos que compiten por ejecutarse (Tiempo Compartido) en dicho procesador.

Esta función la realiza el *planificador de procesos* (proceso 0 ó Scheduler), que además, como veremos más adelante, es el primer proceso que se ejecuta al arrancar el sistema.

Esta *planificación* de CPU de conmutación temporal de procesos, está basada en un sistema de colas FIFO, multinivel con actualización de prioridad.

Gráficamente podríamos representarlo :



- Cuanto más alto sea el número de cola de un proceso menor prioridad tiene ese proceso.
- Los procesos lanzados por los usuarios tienen menos prioridad, su número de cola por defecto es de 10. Pero es posible modificar la prioridad de los procesos externamente (comando **nice**).
- El superusuario puede aumentar la prioridad de los procesos a costa de los demás procesos del sistema que permanecen en cola. Pero, los usuarios solo pueden decrementar la prioridad de los procesos que les pertenecen.

12.3.- PROCESOS DEL SISTEMA

A continuación vamos a presentar la planificación de los principales procesos del sistema UNIX. La planificación más estándar es la siguiente:

Nº DE PROCESO	NOMBRE	FUNCIÓN
0	scheduler ó sched	Planifica el procesador
1	INIT	Inicializador del sistema
2	vhand	Gestor de memoria virtual o swap
3 y 4 (en UNIX 6)	bdflush y bmapflush	Gestor de E/S de disco

En UNIX existen dos tipos de procesos:

Procesos del sistema. Son los procesos que actúan sin que el usuario los solicite. También reciben el nombre de demonios (*daemon*). A su vez, pueden ser de dos tipos:

Procesos permanentes o de larga duración. Se crean cuando se arranca el sistema y permanecen activos hasta que se desconecta. Su función es soportar las actividades del sistema.

Procesos transitorios. Nacen y mueren cuando el sistema efectúa tareas propias, independientes de los usuarios.

Procesos de usuario. Son los procesos asociados a cada usuario como consecuencia de la interpretación de sus órdenes.

Planificador (proceso 0: scheduler): Es el primer proceso en ejecutarse cuando se arranca la máquina. Se encarga de planificar las capacidades de tiempo compartido. Es el responsable de determinar cual de los procesos que están en cola, listos para ser ejecutados, obtienen realmente los recursos de la máquina. También arranca los demás procesos 1 al 6.

Inicializador (proceso 1: init): Se encarga de arrancar y mantener en ejecución los procesos permanentes del sistema, de acuerdo con el contenido del fichero */etc/inittab*.

Gestor de memoria virtual (proceso 2: vhand): Se encarga de realizar la mayor parte del trabajo administrativo del sistema referente a la gestión de la memoria en el entorno multitarea. Es el responsable de gestionar la memoria virtual de la máquina e intercambia procesos activos entre el disco y la memoria principal conforme deban ser ejecutados o aparcados temporalmente. *Sched* y *vhand* trabajan en estrecha relación y componen la parte fundamental del núcleo de UNIX (*kernel*).

Gestores de E/S de disco (procesos 3 y 4: bdflush y bmapflush): Se encargan de gestionar los numerosos buffers (memorias intermedias) de datos en Memoria principal que aumentan la eficacia de las operaciones de E/S con disco, y que funcionan de manera similar a como lo hace un disco de RAM en otros sistemas.

Para evitar pérdidas de información ante un eventual fallo en el sistema, *bdf flush* y *bmap flush* escriben periódicamente todos los buffers generando una operación *sync*, que escribe cada vez aquellos datos que han sido modificados. La frecuencia de esta operación es un parámetro dependiente del sistema, un valor típico es de 20 segundos.

*Existen además otros demonios o procesos del sistema cuya creación y mantenimiento dependen de **init**. Veamos a continuación algunos de ellos:*

Programa de inicialización del sistema (login): Este programa permite solamente lanzar un shell asociado a una cuenta o usuario. UNIX lanza el shell *bash* (uno por cada usuario).

Programa gestor de la línea de terminales (getty): Se encuentra situado en el subdirectorio */etc* y, como se ha indicado, se encarga de gestionar las líneas de los terminales.

El proceso *init* está continuamente enviando *getty* a cada uno de los terminales. En cuanto en un terminal se conecta un usuario (introduciendo su nombre y su password) se pone en marcha el proceso *login*.

Programa planificador del sistema de impresión (lpsched): Se encarga de gestionar el spool de impresión, es decir, del subsistema de impresión, *lp*.

El proceso del sistema (crond): Este proceso es el mecanismo de planificación global o externa del sistema, que actúa, por ejemplo, sobre las órdenes *at* y *batch*. Estas órdenes permiten la temporización y planificación de la ejecución de trabajos.

El demonio *crond* se despierta una vez cada minuto, examina los ficheros de control, que se encuentran en */etc/crontab* según unos sistemas o */etc/spool/cron/crontabs*, según otros. En estos ficheros se almacenan los trabajos planificados mediante *crontab*, si encuentra algún trabajo que deba ser ejecutado en ese minuto los ejecuta y si no los hay, vuelve a dormir hasta el siguiente minuto.

Existe un fichero histórico de todos los trabajos ejecutados mediante *crond*, es el fichero : */usr/adm/cronlog* o en otros sistemas */usr/lib/cron/log*. Esto sucede a partir de la versión 3.0 de UNIX.

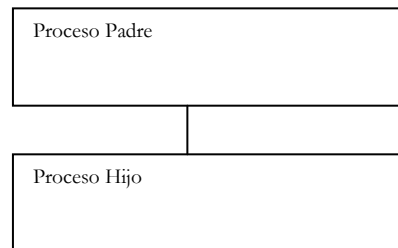
Hay que usar el comando *crontab* para instalar una lista de comandos que se ejecutarán según una planificación regular. La lista de comandos a efectuar en la planificación indicada se ha de incluir en el archivo *crontab*, que se instala con el comando *crontab*. Una vez se ha instalado el archivo *crontab*, mediante el mismo comando, se puede ver la lista de comandos incluidos en el archivo y cancelarla si se quiere.

Antes de instalar el archivo *crontab* con el comando del mismo nombre, hay que crear un archivo que contenga la lista de comandos que se quiere planificar. El comando se ocupa de su colocación. Cada usuario tiene sólo un archivo *crontab* guardado en el directorio */usr/spool/cron/crontabs*.

A este archivo se le asigna el nombre del usuario. Si el nombre de usuario es *usua* y se crea un fichero llamado *micron* y se instala escribiendo *crontab micron*, se creará el archivo *var/spool/cron/crontabs /usua*, con el contenido de micron.

12.4.- PROCESOS SUBORDINADOS

Los procesos describen el estado actual de la máquina. Un proceso es creado desde otro proceso, al cual llamaremos proceso padre.

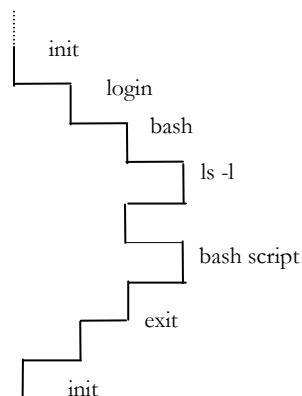


Las órdenes de usuario son procesos subordinados o hijos del proceso shell (bash) correspondiente a ese usuario. Es decir, los procesos lanzados desde un shell dado, son todos procesos subordinados de dicho shell. Al terminar un proceso shell, se terminan automáticamente todos sus procesos.

Cuando se mata un proceso padre normalmente mueren los procesos hijos, aunque pueden seguir activos tomando como nuevo proceso padre el proceso *init*.

Se podría representar mediante el siguiente diagrama de procesos:

Procesos del Sistema



Antes que nada, veamos un ejemplo de los procesos activos que podría haber en un ordenador con sistema UNIX, en un momento dado (para obtener este listado hemos utilizado el comando **ps** que veremos con detalle más adelante). Además, conviene saber que:

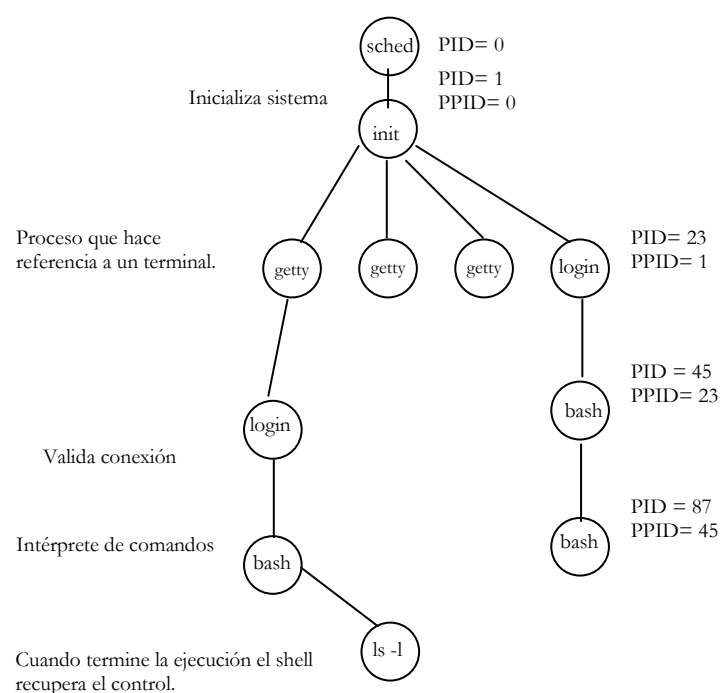
PID: Es el número identificador del proceso.

PPID: Es el número identificador del proceso padre.

\$ps -elf

UID	PID	PPID	C	STIME	TTY	TIME	COMMAND	
R O O T	0	0	0	11:22:04	¿	0:00	sched	planificador
root	1	0	0	11:22:04	¿	0:06	/etc/init	init
root	2	0	0	11:22:04	¿	0:00	vhand	Gestor mem. virtual
root	3	0	0	11:22:04	¿	0:01	bdflush	Vaciado de buffer a disco
root	259	1	0	09:54:35	01	0:02	-sh	shell de root
A101	260	1	2	09:54:35	02	0:01	-sh	shell de ord101
root	232	1	0	09:54:31	¿	0:00	/etc/cron	demonio cron
root	238	1	1	09:54:32	¿	0:02	/usr/lib/lpshed	demonio lp
root	260	1	0	09:54:37	03	0:00	/etc/getty	tty03 m
root	262	1	0	09:54:37	04	0:00	/etc/getty	tty04 m
root	262	1	0	09:54:37	05	0:00	/etc/getty	tty05 m
root	275	1	0	09:54:37	i00	0:00	/etc/getty	ttyi00 m
root	276	1	0	09:54:37	i01	0:00	/etc/getty	ttyi01 m
root	277	1	0	09:54:37	i02	0:00	/etc/getty	ttyi02 m
A101	526	260	11	09:54:38	2	0:00	ps -elaf	comando
A101	527	526	5	09:55:38	02	0:00	ls	comando

Los procesos tienen una estructura jerárquica en árbol, como los directorios,



Veamos un esquema:

Obsérvese que:

Un proceso en marcha genera procesos hijos, cada uno de los cuales se encarga de gestionar distintos aspectos del sistema, y que a su vez, generan otros procesos hijos, hasta llegar a los generados por los usuarios.

El proceso *sched* es padre de *init* pero también lo es de *vhand*, *bdfush*.

A su vez *init* crea y mantiene los procesos permanentes del sistema : *bash*, *cron*, *gettys* y *lpsched*.

Las órdenes de usuario son procesos subordinados o hijos del proceso shell, correspondiente a ese usuario.

Ahora veamos como crea las órdenes de usuario UNIX, convirtiéndolas en procesos. El mecanismo de creación de una orden es el siguiente:

UNIX utiliza un mecanismo de creación por copia.

El proceso hijo es una réplica exacta de su padre.

El proceso hijo hereda la mayoría de atributos de su proceso padre.

UNIX asigna un identificador a cada proceso, denominado PID, en el momento de creación del mismo.

Todo proceso conoce el identificador de su padre, PPID.

La ejecución del hijo es concurrente e independiente. Hay dos formas de crear procesos en el shell:

Primer plano. Espera a que termine para mostrar el prompt.

Segundo plano. También conocido como *background*. El mandato se ejecuta seguido del símbolo **&**. El shell no espera a que el proceso creado finalice, el usuario puede ejecutar otro mandato de forma inmediata. El shell notifica el PID asignado a ese proceso. También notifica al usuario cuando ha terminado de ejecutar el mandato en 2º plano.

Un proceso UNIX dispone de una entrada estándar (**0** asociada al teclado) y dos salidas estándar a la pantalla: una normal (**1**), y otra de errores (**2**). Estas salidas y entradas son heredadas por los procesos.

12.5.- DIAGNÓSTICO DE PROBLEMAS CON PROCESOS Y SOLUCIÓN.

Conviene conocer aproximadamente cuales son los procesos que habitualmente se ejecutan y las características que tienen, estos varían entre las

diferentes versiones de UNIX (por instalación , por configuración, por opciones del sistema, etc)

Para conocer estos procesos conviene ejecutar con frecuencia la orden **ps**, esto permite detectar si algún proceso no se ejecuta o se ejecuta incorrectamente.

La mayor parte de los problemas están relacionados con la ejecución de una orden o aplicación concreta y frecuentemente se trata de :

- Falta de ficheros o directorios, porque se han borrado o deteriorado.
- Permisos inadecuados en ficheros o directorios.

Las claves para detectar los problemas relacionados con procesos son las siguientes:

- Una modificación notable en el tiempo de respuesta, y por tanto, una baja en el rendimiento del sistema.
- Terminación prematura de procesos sin causa aparente.
- Actividad inusual del disco sin causa aparente.

UNIX dispone de herramientas para eliminar o matar procesos largos o que no se ejecutan correctamente (comando **kill**).

Si elimina un proceso, conviene recordar que, se eliminan al mismo tiempo todos sus procesos hijos o subordinados activos. Por ejemplo al eliminar un *shell*, se eliminan todos los procesos lanzados por él.

En UNIX a veces esto no sucede, deberemos asegurarnos de que los procesos hijos se han eliminado y si no lo tendremos que hacer nosotros.

El superusuario puede eliminar cualquier proceso del sistema o de usuario, excepto los procesos 0,1,2,3 y 4.

Los usuarios solo pueden matar los procesos propios o los de aquellos usuarios no protegidos. En UNIX, es posible que esta protección no esté activa y por tanto deberá ser el superusuario el que tome precauciones, ya que la eliminación de alguno de estos procesos puede dañar el sistema.

12.6.- FIN DE SESIÓN DE TRABAJO CON PROCESOS EN MARCHA

Normalmente, los subordinados de un proceso finalizan cuando el superior muere o finaliza. Esto significa que un proceso, iniciado en segundo plano finaliza cuando se desconecta del sistema. Bien por salida natural, *exit* o *^D*, bien porque ha sido matado.

UNIX proporciona una herramienta para permitir que los procesos subordinados a un shell de usuario, continúen, ejecutándose después de la despedida de la sesión de usuario. Esta herramienta es el comando **nohup**.

Esta posibilidad es muy útil para trabajos que necesiten ejecutarse durante un largo periodo de tiempo, durante toda una noche e incluso durante días y semanas.

La despedida del shell mientras procesos subordinados están en marcha permite la protección del proceso, ante fallos en el sistema, o ante investigaciones en el shell de un usuario que perturbe la confidencialidad de la información almacenada en su entorno de trabajo.

A nivel interno, esta herramienta funciona porque el proceso subordinado que se desea continúe, cuando la sesión de usuario ha finalizado, toma como padre o líder el proceso *init* (PID 1) y no el proceso *bash* correspondiente a ese usuario.

12.7.- COMANDOS

ps

Esta orden, muestra la información de los procesos. Este informe es solamente como una instantánea de lo que está pasando cuando ha preguntado. Una invocación subsecuente puede dar resultados distintos.

Sintaxis

ps [-] [*opciones*]

Por ejemplo:

```
Remigio:~$ ps
  PID TTY STAT  TIME COMMAND
 2500 p1 S    0:00 -bash
 2897 p1 R    0:00 ps
```

Modificador	Resultado
l	Formato largo: muestra más información sobre cada proceso.
u	Muestra información sobre el usuario propietario del proceso y el tiempo de ejecución.
s	Muestra información sobre las señales que maneja cada proceso.
m	Muestra información sobre las memoria que maneja cada proceso.
a	Muestra también procesos de otros usuarios.
h	Elimina la línea de encabezado (con los títulos de las columnas).
r	Sólo muestra los procesos que se están ejecutando (no los parados).
j	Produce la salida en el formato del trabajo.
f	Información más completa de cada proceso, con el usuario propietario de cada proceso, tiempo del comienzo de la ejecución, etc.
x	Muestra todos los procesos, incluidos demonios.
c	Lista el nombre del comando desde la estructura de tareas del kernel
e	Muestra el entorno.
w	Visualiza en formato ancho.
n	Proporciona salida numérica para USER y WCHAN
txx	Sólo muestra procesos con los controles tty xx

El informe muestra información en columnas estas son:

PID	El número identificador de proceso.
PPID	El número identificador del proceso padre.
PRI	Prioridad de proceso.
NI	El valor del <i>nice</i> del proceso. Un valor positivo significa menos tiempo de CPU.
SIZE	El tamaño virtual de la imagen, calculado como el tamaño de texto+pila+datos.
RSS	El tamaño del conjunto residente. El número de Kilobytes del programa que está residente en la memoria actualmente.
WCHAN	El número del evento del kernel por el que está esperando el proceso.
STAT	El estado del proceso. Dado por uno de los códigos siguientes: R Ejecutable S Dormido D Dormido ininterrumpible. T Parado o rastreado. Z Zombie. Es un proceso terminado, y el superior no lo ha asumido. W El proceso no tiene páginas residentes.
TT	El nombre del tty de control para el proceso.
PAGEIN	El número de fallos de página que han causado que las páginas se lean desde el disco.
TRS	El tamaño de texto residente.
SWAP	El número de Kilobytes de un espacio intercambiable

Por ejemplo,

ps -uaxl Muestra la información más completa. Si observamos los valores PID y PPID se podrá seguir la secuencia de creación de procesos hijos por parte de sus líderes.

kill

El comando **kill** sirve para enviar señales a procesos. Si el proceso no está preparado para recibir la señal, terminará inmediatamente su ejecución (morirá). La señal 9 (SIGKILL) siempre mata al proceso. Normalmente **kill** se emplea para matar procesos, de ahí su nombre.

Sintaxis

kill [*opciones*] procesos

Los procesos se identifican mediante su *PID*. Para averiguar el PID de un proceso tenemos varias opciones:

- Utilizar el comando **ps**.

- Al ejecutar un comando en segundo plano (poniendo `&` en la línea de comandos), el intérprete de comandos nos indica el *PID* del proceso que queda en segundo plano.

La principal opción que podemos utilizar es `-s`, que nos permite a continuación especificar el número de señal a enviar (mediante su nombre –en mayúsculas– o su número). Para ver las señales disponibles podemos utilizar la opción `-l`:

```
Remigio:~$ kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL
5) SIGTRAP     6) SIGIOT     7) SIGBUS      8) SIGFPE
9) SIGKILL     10) SIGUSR1   11) SIGSEGV    12) SIGUSR2
13) SIGPIPE    14) SIGALRM   15) SIGTERM    17) SIGCHLD
18) SIGCONT    19) SIGSTOP   20) SIGTSTP    21) SIGTTIN
22) SIGTTOU    23) SIGURG    24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH   29) SIGIO
30) SIGPWR
```

Por defecto `kill` utiliza la señal `SIGTERM`, habitualmente utilizada para abortar procesos. Como `-s` es la opción más habitual, también podemos omitirla, escribiendo el nombre o número de la señal precedido de un guión. Los siguientes ejemplos envían la señal de muerte al proceso cuyo *PID* es el 2543 (todos los comandos son equivalentes):

```
kill -9 2543
kill -s 9 2543
kill -SIGKILL 2543
kill -s SIGKILL 2543
```

Un usuario normal sólo puede enviar señales a procesos que sean de su propiedad (comandos que él mismo haya ejecutado).

La tabla siguiente contiene las principales opciones disponibles para este comando.

Modificador	Resultado
<code>-l</code>	Muestra un listado de los nombres de señal y sus números.
<code>-XXXXX</code>	<code>-XXXXX</code> es el nombre o número de señal que hay que enviar.
<code>-s</code>	Permite especificar a continuación especificar el nombre o número de señal que hay que enviar.

A continuación se muestran las señales que se pueden enviar:

Modificador	Resultado
1	Provoca que el terminal queda colgado. Al salir del sistema los procesos entran en modo desatendido o background (nos permite cortar un proceso de shell, salir al login) (SIGHUP similar a colgar el teléfono)
2	Interrupción similar a Ctrl-c
3	Salir y generar un archivo de volcado <i>core</i>
9	Kill o muerte segura. No puede ser ignorada (SIGKILL)
15	Finalización del proceso. Es similar al 9 pero puede ser ignorada. Predeterminada (SIGTERM)

El **PID** es el identificador del proceso al que quiere enviar la señal especificada. Un *pid* es un número utilizado por el sistema para hacer un seguimiento del proceso. Para averiguar esto, como antes se ha explicado, se utiliza el comando *ps*.

crontab

El comando *crontab* informa al proceso daemon *crond* de los programas y de la planificación según la que deberían ejecutarse.

<i>Sintaxis</i>
crontab [- <i>opciones</i>] [-u <i>usuario</i>]

Esta orden admite los siguientes parámetros:

Modificador	Resultado
-l	Lista el archivo <i>crontab</i> .
-e	Edita el archivo <i>crontab</i> . Utiliza el editor <i>vi</i> o el especificado por su variable de entorno VISUAL.
-r	Suprime el archivo <i>crontab</i> del usuario.
-u <i>usuario</i>	Especifica un archivo <i>crontab</i> determinado del usuario que se va a manipular. Tiene que ser <i>root</i> para utilizar esta función.

El archivo *crontab* tiene el formato : *M H D m d comando*. Se ignoran las líneas en blanco y aquellas que comienzan por #. Un * en un campo selecciona todos los valores posibles de ese campo. Además, el archivo *crontab* puede especificar como campos del comando rangos .

Formato:

M Los minutos de la hora. (0-59)

H	La hora del día. (0-23)
D	El día del mes. (1-31)
m	El mes del año. (1-12)
d	El día de la semana. (0-6, 0=domingo)
comando	El programa a ejecutar. Esta línea de comandos se pasa a sh . El shell se ejecuta con sólo tres variables de entorno: USER, HOME, SHELL.

Cada línea del archivo *crontab* contiene un patrón de tiempo y un comando; según aquel se ejecuta este. El patrón se divide en cinco campos separados por espacios o tabuladores. Cualquier salida que no aparezca en la estándar se dirige al usuario, mediante el correo.

Para que un usuario pueda usar el comando *crontab*, ha de estar listado en el archivo */etc/cron.d/cron.allow* (A partir de la versión 3.0). El superusuario debe añadir el nombre del usuario que se desea habilitar para usar este comando. Una vez creado el archivo *crontab*, hay que modificarlo a través del comando *crontab*. No se debe modificar por ningún otro medio.

Se pueden disponer de tantas entradas como se quieran en un archivo *crontab* y señalarlas para que se ejecuten en cualquier momento.

Una vez creado el archivo de *crontab* para poder añadir entradas, para corregirlas o para eliminarlas tendremos que editar el fichero de *crontab* y modificarlo según nuestras necesidades.

Ejemplo,

```
30 07 * * 01 sort fichero_ventas | mail "ventas semanales" usuario
```

Se va a clasificar el fichero de ventas y el resultado se va a enviar mediante un mail a un usuario concreto, la clasificación se va producir el minuto 30 de la hora 7 de cualquier día del mes y cualquier año, un lunes.

```
*/2 * * * * echo Hola han pasado dos minutos.
```

Aparecerá el mensaje cada dos minutos.

```
10-20/2 * * * * echo Hola entre los minutos 10 y 20 aparezco cada dos minutos
```

El mensaje aparecerá entre los minutos 10 al 20 de cada hora y con un intervalo de dos minutos.

```
10,15 * * * * echo Hola es el minuto 10 ó 15
```

El mensaje aparece los minutos 10 y 15 de cada hora

Todas estas combinaciones se pueden hacer con el resto de apartados.

at

El comando *at* planifica trabajos para que se ejecuten posteriormente, en un momento dado.

El superusuario tiene un fichero *crontab* en el cual una de sus entradas puede ser parecida a la siguiente:

```
* * * * /usr/lib/atrun 1> /dev/null 2> /dev/null
```

lo cual significa que cada minuto se debe ejecutar *atrun* que es el encargado de que se ejecute el comando *at*.

Sintaxis

at [-*opciones*] fecha-hora

El comando *at* devuelve un identificador de trabajo cuando se le invoca. Este identificador puede ser utilizado luego como el parámetro *id-trabajos*.

Esta orden permite las siguientes opciones:

Modificador	Resultado
-q cola	Se puede especificar un nombre de cola opcional, cola puede ser cualquier letra entre la <i>a</i> y <i>z</i> , y entre la <i>A</i> y <i>Z</i> . La cola <i>c</i> la predeterminada para at y la cola <i>E</i> lo es para <i>batch</i> . En tiempo de CPU, las colas con letras más altas (más cercanas al final del alfabeto) se ejecutan con prioridad más baja. Si se envía un trabajo a una cola nombrada con una letra mayúscula, se trata como si la hubiera enviado <i>batch</i> .
Fecha - hora	Es la hora en que empieza el trabajo. El formato es altamente flexible y se divide en tres partes básicas : hora, fecha e incremento. Se acepta la hora en formato HHMM o HH:MM. También se puede especificar el sufijo AM o PM y el día en que se va a ejecutar el comando dándole la fecha con el formato MMDDAA o MM/DD/AA o MM.DD.AA. También se puede indicar a at que ejecute el trabajo hoy o mañana, poniendo <i>today</i> o <i>tomorrow</i> .
-l	Lista los trabajos actualmente planificados. Si es superusuario, se listan todos los trabajos.
-m	Envía correo al usuario cuando el trabajo se ha completado, incluso si no hay ninguna salida.
-d idtrabajo	Elimina de la cola los identificadores de trabajo. Solo se pueden eliminar los propios trabajos, a no ser que sea el superusuario.
-f archivo	Lee el trabajo desde el fichero especificado en lugar de la entrada estándar.
-v	Visualiza la versión del at
-V	Se utiliza con la opción '-l' y visualiza los trabajos que se han realizado pero no borrado.

Dentro de esta orden podemos distinguir las siguientes sub-órdenes:

atq: Lista los trabajos pendientes del usuario, (**at -l**). Permite utilizar la opción ‘-v’

atq

atq -v

atrm: Borra un trabajo. (**at -d n°trabajo**)

atrm n°trabajo

Directorios usados

/etc

El superusuario puede utilizar este comando en cualquier momento. Para los otros usuarios, el permiso para utilizar **at** lo determinan los archivos */etc/at.allow* y */etc/at.deny*.

Si no existe *at.allow* y existe *at.deny*, pero éste está vacío, todos los usuarios podrán utilizarlo.

Si no existe ninguno no podrá utilizarlo ningún usuario, (sí, el superusuario)

Si existe el fichero *at.allow* debe contener el nombre de los usuarios que pueden utilizar el comando (uno por línea). Da lo mismo que exista o no el *at.deny*.

Si no existe *at.allow*, entonces los nombres que se introduzcan en *at.deny* serán los usuarios que no puedan utilizar el comando, todos los demás podrán utilizarlo.

/var/spool/atjobs

En este directorio va creando unos ficheros con nombre de 14 caracteres, que son los trabajos que vamos mandando.

Los ficheros los crea ejecutables y una vez que se han ejecutado les quita el permiso de ejecución.

De estos 14 caracteres que forma el nombre del fichero el significado de los 6 primeros caracteres es el siguiente: (empezando por la izquierda)

1º Es el nivel, la prioridad que le demos con la opción ‘-q’.

2º al 6º Es el número de trabajo en hexadecimal.

La salida estándar y los errores estándar de los comandos ejecutados se envían por correo al usuario por medio del *mail* de Unix..

Ejemplos,

at 10:00am tomorrow <fichero>

at -f fichero 11am Aug 15

De esta forma los comandos no son introducidos a continuación sino que se extraen del fichero especificado.

Ejecuta los comandos encontrados en el fichero especificado el 15 de Agosto a las 11:00 AM.

batch

El comando *batch* planifica trabajos para que se ejecuten posteriormente, a diferencia de *at* es el sistema quien decide cuando ejecuta el comando. El comando *batch* deja que sea el sistema operativo el que decida el momento adecuado para ejecutar el proceso. Cuando se planifica una tarea con *batch*, UNIX comienza y trabaja en el proceso siempre que la carga no sea demasiada alta. Las tareas que se ejecutan con *batch* se hacen en segundo plano, como *at*.

Sintaxis

batch [INTRO]

//Lista de órdenes

[ctrl.+d] ^D

Se puede almacenar la lista de comandos en un archivo y redirigir la entrada a *batch* para que provenga del archivo.

Ejemplo,

```
batch <intro>
sort /usr/reports |lp
echo "Ficheros impresos " |mail usuarios
^D
```

El sistema devuelve la siguiente respuesta :

job 7789001234 at Sat Dec 21 11:43:09 1996

La fecha y hora listadas son las del momento de pulsar $\wedge D$ para completar el comando *batch*. Cuando la tarea esté completa se debe consultar el correo porque es ahí donde se dirigen los mensajes.

nice

Sirve para modificar la prioridad de un proceso.

Sintaxis

nice –número | - - número comando

Este comando da un control sobre la prioridad de una tarea respecto a otras. Si no se usa *nice* los procesos se ejecutan con una prioridad definida. Se puede disminuir (de forma amable como la traducción literal del comando indica) la prioridad de un proceso con este comando de forma que los otros procesos puedan planificarse para que usen más frecuentemente la CPU. El superusuario también puede aumentar la prioridad de un proceso.

El nivel de prioridad lo determina el argumento *número* (un número más alto significa una prioridad menor). El valor predeterminado, como se ha mencionado, es **10**. Si el argumento número se ha especificado, la prioridad se incrementa en esta cantidad hasta un límite de 20. Si no se especifica nada todos los procesos toman la prioridad 10.

Para ver la prioridad *nice* de los procesos se usa la opción *-l* del comando **ps**. Hay que observar las columnas PRI y NI.

Para dar a un proceso la mínima prioridad se especifica en el argumento número el 10.

Sólo el superusuario puede aumentar la prioridad de un proceso. Para ello se usa un número negativo como argumento de *nice*. Se recuerda que cuanto más bajo sea el valor de nice más alta será la prioridad.

Ejemplos,

```
nice -5 lp mis_listados &  
nice -10  
  
nice - -10 ls -lR /
```

Baja la prioridad , es decir $10+5=15$ en la cola.
Esto da la mínima prioridad posible. El número de cola será $10+10=20$
Sólo el superusuario. Da la máxima prioridad $10-10=0$

Por último conviene mencionar que en algunos sistemas existe un comando, **renice**, que permite modificar la prioridad de un proceso que ya se está ejecutando. Los sistemas Unix de Berkeley lo incorporan y también los sistema Linux system V. La sintaxis es similar a la de **nice**.

nohup

Mantiene la ejecución de ordenes aunque se desconecte el sistema.

Sintaxis

nohup orden1;orden2;... ordenN

El efecto que produce es ejecutar una orden aunque finalice su líder, es decir cambia el PPID de ese proceso asignándole el valor 1.

Para la ejecución de **nohup** deben tenerse en cuenta las siguientes consideraciones:

Si después de la orden no está disponible el prompt del entorno de usuario, no se podrá despedir la sesión. Por ello, es conveniente utilizar la ejecución desatendida en la orden argumento de **nohup**.

Otra consideración a tener en cuenta es que si no se redirige la salida de la orden, por defecto, crea un fichero con nombre: **nohup.out** dónde se almacena el resultado de la orden.

El Linux genera errores que graba en ficheros llamados **core**, los cuales ocupan mucho espacio. Conviene ver el contenido de estos ficheros y después borrarlos.

Para borrar estos ficheros lo primero es localizarlos. Esto lo haremos con la orden:

find / -name core -print

Este comando buscará todos los ficheros **core**, y los escribirá en la pantalla. Después iremos a cada directorio donde estén y los borraremos.

&

Ejecuta la orden o proceso precedente en segundo plano. El shell devuelve el control al terminal mientras ejecuta el proceso en segundo plano. Cuando ejecutamos el proceso, se nos devuelve un número. Se trata del indicativo del proceso, que podremos usar para detenerlo o matarlo, por ejemplo.

\$

Variable de entorno generada por el shell que contiene el PID del proceso en ejecución. Su valor puede visualizarse con la orden **echo \$\$**.

!

Variable de entorno generada por el shell, que contiene el PPID del proceso en ejecución. Su valor puede visualizarse con la orden **echo \$!**.

time

El comando `time` sirve para medir el tiempo de ejecución de otros comandos. Admite como parámetro otro comando. `time` ejecuta ese comando y después nos imprime el tiempo que ha tardado. Este tiempo se separa en tiempo de usuario, tiempo de núcleo, fallos de caché, etc.

Sintaxis

time [*opciones*] comando [*argumento del comando*]

Por ejemplo:

```
Remigio:~$ time sleep 2
0.00user 0.00system 0:02.00elapsed 0%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (62major+8minor)pagefaults 0swaps
```

Supr

Tecla o carácter de interrupción usado para abortar un proceso durante su ejecución. Se encuentra definido en la configuración de la orden **stty** (que veremos más adelante). Puede visualizarse en el campo *intr* que aparece en la orden *stty -a*. Puede cambiarse con la orden *stty intr [tecla]*.

Aquí, [tecla] es la tecla o combinación de teclas que generará, como por ejemplo ctrl.+Z, Pausa, o la que queramos. Se puede paralizar una orden temporalmente con la orden **stty susp [tecla]**.

sleep

El comando `sleep` sirve para hacer una pausa (detener la ejecución) durante un tiempo (inicialmente especificado en segundos).

Sintaxis

sleep tiempo [unidad de tiempo]

Esta orden puede errar en su temporización hasta 1 segundo, de modo que la orden

```
$ sleep 1
```

puede no producir resultados exactos.

Uno de los usos más sencillos de **sleep** es crear un reloj de alarma. Esta orden hará sonar la campanilla del terminal y mostrará un mensaje 10 minutos después de introducirse la orden.

```
$ sleep 600; echo "\007 Hora de comer! \007" &
```

Esta orden funcionará solamente si no nos despedimos de la máquina hasta después de que se desactive la alarma, pero como la orden se ha ejecutado en modo subordinado con **&** podemos continuar haciendo otros trabajos mientras se está ejecutando. La cadena `\007` de la orden **echo** es otra forma de especificar caracteres para eco: `007` es el código ASCII octal de la combinación de tecla [ctrl.+G], que hace sonar un pitido en el terminal. La cadena `\007` en este contexto es muy diferente de `\07`, `007`, `7` ó `\7`. Al escaparlo con la diagonal inversa estamos diciendo a **echo** que éste es un carácter único en lugar de tres dígitos.

Ya que el sistema UNIX es multitarea, deberíamos ceder el control de la CPU cuando deseemos esperar durante un intervalo. Esto permite a otro programa trabajar mientras nuestra orden está demorándose. Son habituales las iteraciones de cuenta de algún bucle activo, pero este planteamiento de *espera activa* puede ser muy despilfarrador en el sistema UNIX, donde otros programas pueden estar demandando el uso del sistema mientras nuestra orden está haciendo tiempo. Un beneficio de la orden **sleep** es que su temporización depende del reloj interno y no de la carga del sistema o de la velocidad básica del procesador.

El tiempo es un número entero. La unidad de tiempo es alguno de los siguientes caracteres: s (segundos), m (minutos), h (horas), d (días). Si se omite la unidad de tiempo se presupone s (segundos).

Ejemplo,

hacer una pausa de 10 segundos: **sleep 10**

jobs

Muestra los procesos activos en el sistema.

fg

Si se han ejecutado varios procesos en segundo plano desde el shell, se pueden volver a presentarlos en primer plano con ayuda del comando **fg**. Igual que el comando **bg**, al comando **fg** puede incorporársele el número de trabajo como parámetro.

Sintaxis

fg [*número de trabajo*]

En el shell, el número de trabajo aparece tras un signo de porcentaje. El número de trabajo de un proceso en segundo plano puede averiguarse mediante el comando **jobs** (visto anteriormente). Estos son los posibles parámetros:

Modificador	Resultado
%Número	El proceso con el número indicado.
%Texto	El proceso cuyas líneas de comandos empiezan por el texto.
%?Texto	El proceso cuyas líneas de comandos contienen el texto en alguna parte.
%%	El proceso actual (o también %+)
%-	El proceso anterior.

Si no se especifica ningún parámetro, el proceso actual (el último proceso en segundo plano que se ha ejecutado) se pasa al primer plano. El proceso en curso puede volver a detenerse con la combinación de teclas [ctrl.+z] (veremos el comando **stty** más adelante).

bg

Con ayuda de este comando se seguirá ejecutando un proceso de forma asincrónica dentro del shell. Cada comando en segundo plano del shell puede identificarse de forma unívoca mediante un número de trabajo.

<i>Sintaxis</i>
bg [<i>Número de trabajo</i>]

Los comandos en segundo plano (proceso asincrónico) pueden pasarse a primer plano (comando **fg**, visto anteriormente). Los comandos en primer plano se detendrán mediante la señal **SIGSTOP** (las señales se verán a continuación):

Finalmente, con el comando **bg** puede volver a convertirlos en comandos en segundo plano.

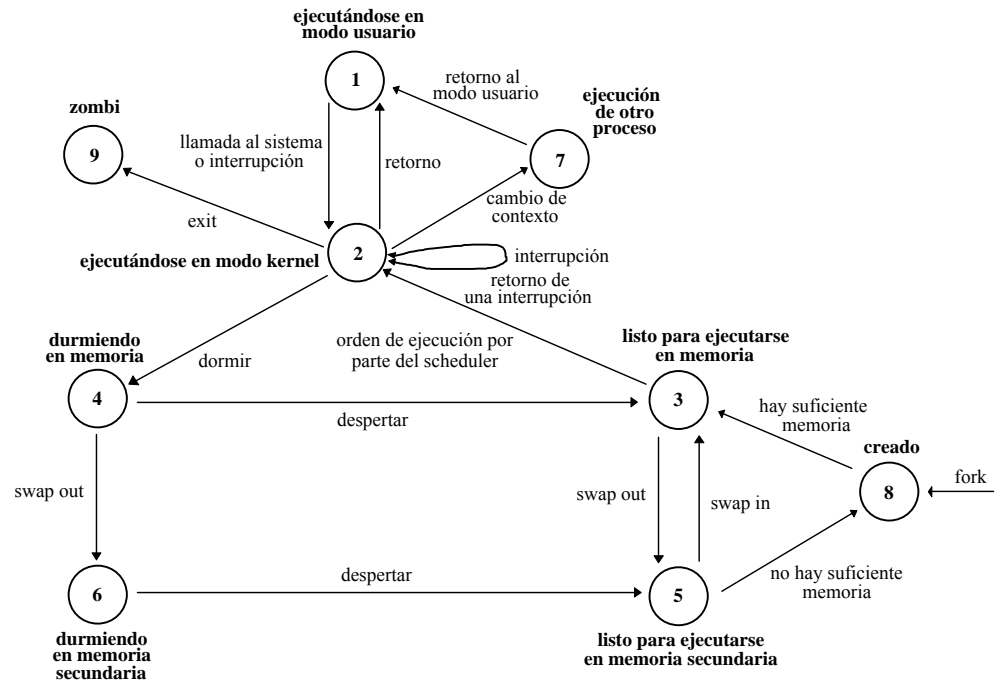
Como parámetros de este comando puede usarse una de las siguientes indicaciones para seguir procesando el comando deseado de forma sincrónica:

Modificador	Resultado
%Número	El proceso con el número de trabajo indicado.
%Texto	El proceso cuyas líneas de comandos empiezan por el texto.
%?Texto	El proceso cuyas líneas de comandos contienen el texto en alguna parte.
%%	El proceso actual (o también %+)
%-	El proceso anterior.

Si no se indica ningún parámetro, el proceso actual pasará a ser un proceso en segundo plano.

12.8.- LLAMADAS AL SISTEMA. *FORK, EXEC Y WAIT*

Estados de un proceso:



12.8.1.- Familia de llamadas *exec*

Toda la familia de llamadas *exec* permiten transferir el control del proceso desde el programa actual a uno nuevo. En cualquiera de sus formatos, una llamada a *exec* superpone la imagen del proceso nuevo sobre la del antiguo lo que implica que si una llamada *exec* se realiza sin errores no se puede regresar al proceso inicial.

Exec sustituye el segmento de código del proceso original con el del nuevo y crea un nuevo segmento de datos. Los descriptores de ficheros abiertos por el programa inicial permanecen abiertos en el nuevo, con la excepción de aquellos que tengan activo el bit *close_on_exec*.

Las zonas de memoria compartidas del programa antiguo no se conectan al nuevo programa. Los identificadores reales de usuario y de grupo del proceso no se cambian, sin embargo los efectivos pueden cambiar si están activos los bits *set_user_id* o *set_group_id*.

Una llamada a *exec* sólo vuelve al proceso de llamada en el caso de producirse algún error, además, en este caso, la llamada devuelve el valor -1.

La diferencia entre las distintas llamadas a *exec* está en :

- Formato de su argumentos
- Entorno de trabajo
- La variable PATH

De manera que todas aquellas llamadas que no exportan el entorno de trabajo de manera automática finalizan con la letra “e” (*execle*, *execve*....) y aquellas que realizan una búsqueda automática por el PATH finalizan con la letra “p” (*execlp*, *execvp*....).

Las funciones son (<unistd.h>) :

- int **execv** (const char *path, const char *argv[])
Ejecuta el programa indicado por *path* como un nuevo proceso. El parámetro *argv[]* es un array de cadenas terminadas en 0 que se usa para dar valor al parámetro *argv* de la función *main()* del programa que se va a ejecutar. El último elemento de este array debe ser un puntero nulo. Por convención el primer elemento de *argv* es el nombre del fichero. El entorno se exporta automáticamente tomándolo de la variable *environ* del proceso actual. No se realiza búsqueda automática por el path, por lo que habrá que especificar el nombre completo del fichero a ejecutar.
- int **execl** (const char *path, const char *arg0,....., const char *argn)
Igual que *execv* sólo que las cadenas de *argv* se especifican individualmente. El último de los argumentos debe ser un puntero nulo.
- int **execve** (const char *path, const char *argv[], const char *env[])
Igual que *execv* sólo que el entorno no se exporta automáticamente y se puede especificar explícitamente en el parámetro *env[]*. Es un array de cadenas con la misma forma que la variable *environ* (nombre=valor).
- int **execle** (const char *path, const char *arg0,...,const char *argn,const char *env[])
Similar a *execl* pero permite especificar el entorno para el nuevo proceso explícitamente.
- int **execvp** (const char *fich, const char *argv[])
Igual que *execv* sólo que realiza la búsqueda automática por le PATH.
- int **execlp** (const char *fich, const char *arg0,....., const char argn)
Similar a *execl* con la excepción de que se realiza una búsqueda automática por el PATH. El entorno se exporta automáticamente.

El programa llamado con *exec* heredará el *user área* del proceso que realiza la llamada por lo tanto recibirá: el PID, PGID y TGID (terminal group id).

12.8.2.- Llamada *fork*

Permite generar un nuevo proceso o proceso hijo es una copia exacta del proceso padre. El formato de la llamada *fork()* es :

```
#include <sys/types.h>
#include <unistd.h>
```

pid_t fork()

Si la llamada tiene éxito habrá dos procesos, el padre y el hijo, que verán el retorno de la llamada pero con un valor diferente para cada uno de ellos. Retornará 0 para el proceso hijo y el identificador del proceso hijo para el padre. Si se produce un error retornará -1 al padre.

El proceso hijo hereda :

- Los id real y efectivo de usuario y grupo
- El entorno del proceso (las variables que hayan sido exportadas)
- Los valores para la manipulación de señales
- La clase del proceso
- Los segmentos de memoria compartida
- El directorio root y el actual
- La máscara de creación de ficheros
- Los límites de recursos y el terminal de control asociado.

El proceso hijo difiere en :

- El hijo tiene un ID de proceso único
- Dispone de una copia privada de los descriptores de ficheros abiertos por el padre
- El conjunto de señales pendientes del proceso hijo es vaciado
- El hijo no hereda los bloqueos de ficheros establecidos por el padre

12.8.3.- Llamada a *exit*

Un proceso puede terminar su ejecución mediante una llamada a *exit()* o al recibir una señal o al finalizar su función *main()*. El formato de *exit()* es el siguiente :

```
#include <stdlib.h>
void exit (int status)
```

Finaliza el proceso liberando los recursos que estaba utilizando : libera áreas de memoria, cierra los ficheros y se autoelimina de la tabla de procesos. El valor de *status* es pasado a la llamada *wait()* del proceso padre y los valores pueden ser :

- ☞ 0 Todo es correcto
- ☞ 1-255 Indican códigos de error definidos por el usuario

12.8.4.- Llamada *wait*, *waitid* y *waitpid*

Wait() suspende la ejecución que la emplea hasta que uno de los hijos directos terminan o bien hasta que un hijo que está siendo trazado se detenga al recibir una señal.

```
#include <sys/types.h>
#include <sys/wait.h>
```

pid_t wait (int *status)

Retorna el identificador del proceso que finalizó. En caso de error retorna -1. En *status* se almacenará la siguiente información :

- Si el proceso hijo que está siendo trazado ha sido detenido, los 8 bits de mayor peso contendrán el número de la señal que provocó la parada, los otros 8 bits serán iguales o tendrán el valor WSTOPFLG.
- Si el proceso hijo terminó debido a una llamada a *exit()* los 8 bits de mayor peso contendrán los 8 bits de menor peso del argumento en la llamada a *exit()*.
- Si el proceso hijo terminó debido a la recepción de una señal, los 8 bits de mayor peso estarán a 0 y los otros 8 tendrán el número de la señal recibida.

La llamada *waitid()* permite que un proceso sea suspendido hasta que un proceso hijo o un grupo de procesos cambie de estado. El formato es el siguiente :

int waitid (idtype_t idtype, id_t id, siginfo_t *infop, int op)

Waitid() retorna 0 si el proceso o los procesos hijos han terminado o si han cambiado de estado y -1 en caso de error.

Idtype indica el tipo del argumento *id*, los posibles valores son :

- ☞ P_PID : Indica que hay que esperar por el proceso cuyo pid coincida con el valor de id
- ☞ P_GID : Espera por todos los procesos del grupo id
- ☞ P_ALL : Espera por todos los procesos del grupo

En ****infop*** se encuentra el estado del proceso hijo. La estructura *siginfo_t* tiene los siguientes campos :

- ☞ int si_signo : Número de señal
- ☞ int si_errno : Número de error

☞ `int si_code`: Información sobre el motivo de la señal

Op es la combinación de los siguientes flags :

☞ **WEXITED** Indica que hay que esperar que el proceso o los procesos terminen

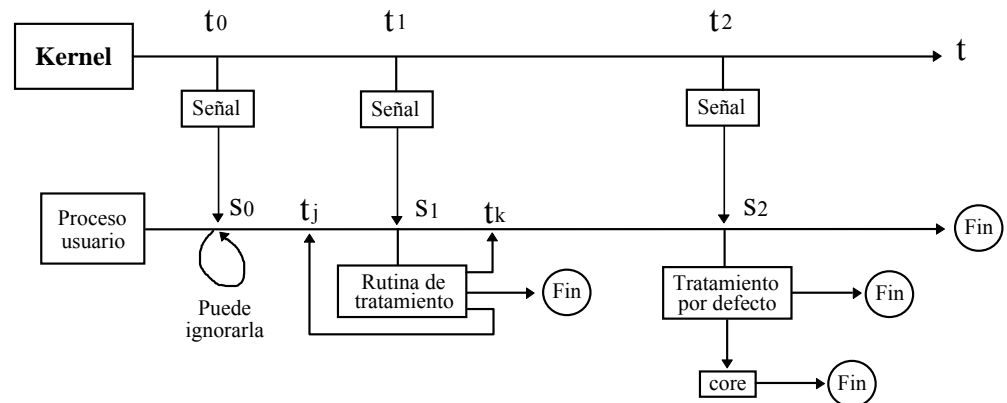
☞ **WSTOPPED** Hay que esperar que el proceso o los procesos sean detenidos y devolver en *info* información sobre la señal

☞ **WNOHANG** El proceso debe retornar inmediatamente, no tiene que pasar a sep

☞ **WNOWAIT** Para obtener información sobre el proceso hijo en *info* sin liberar la estructura de proceso de ese proceso.

12.9.- SEÑALES

Una señal es un mensaje de un proceso a otro con relación de parentesco. Estas señales software se envían en un instante determinado pero el proceso no sabe cuando le llega, ni quien se la envía, pero si sabe que interrupción le llega.



El Unix SVR4 pone a disposición del usuario 31 interrupciones (la int. 32 está a disposición de cada fabricante).

Las interrupciones 16 y 17 son de usuario y los procesos las pueden utilizar para comunicarse. Se agrupan en cinco categorías :

▪ **Condiciones hardware : Enviadas por el sistema cuando se produce un error durante la ejecución de un proceso.**

☞ **SIGSEV** : Será enviada a un proceso cuando acceda a direcciones que están fuera de su espacio de direccionamiento o a zonas de memoria protegidas.

☞ **SIGBUS** : Cuando se intenta realizar un acceso a un hardware sobre el que no tengo privilegio.

☞ **SIGILL** : Ejecución de una instrucción ilegal.

▪ **Condiciones software : Señales generadas por un requerimiento del usuario.**

☞ **SIGINT**: Generada cuando se pulsa *ctrl-c* (es enmascarable).

☞ **SIGQUIT**: Generada cuando se pulsa *ctrl-esc*

☞ **SIGTERM**: La envía un usuario a un proceso con la orden kill. El proceso morirá si no está en modo sistema. Si lo está se esperará a que termine.

☞ **SIGHUP**: Indica un corte en la línea de comunicaciones (apagas el terminal).

☞ **SIGKILL**: Indica muerte imperativa. No es enmascarable.

☞ **SIGUSR1**: Definibles por el usuario.

☞ **SIGALRM**: Es enviada a un proceso cuando alguno de sus temporizadores descendientes llega cero.

▪ **Que efectúan notificaciones de operaciones de E/S**

☞ **SIGPOLL** : La revive un proceso cuando se está produciendo una e/s por un stream que está siendo gestionado por una interrupción poll.

▪ **Control de procesos**

☞ **SIGSTOP** : Saca a un proceso de la zona de swapping (sólo en SVR4). Señal de parada de un proceso.

☞ **SIGCONT**: Lleva al proceso a background. Continúa la ejecución del proceso.

☞ **SIGCLD** : El hijo la envía la padre cuando cambia de estado.

▪ **Control de recursos**

☞ **SIGXCPU**: Exceso de consumo de CPU o de memoria por parte del proceso.

☞ **SIGXFSZ** : El proceso ha sobrepasado el tamaño máximo para un archivo.

12.9.1.- Tipos de señales

Número	Señal
1	Hangup
2	Interrupt
3	Quit
4	Illegal instruction
5	Trace trap
6	IOT instruction
7	Floating point exception
8	EMT instruction
9	Kill
10	Bus error
11	Segmentation violation
12	Bad system call argument
13	Write on unread pipe
14	Alarma clock
15	Software termination signal
16	User defined signal 1
17	User defined signal 2

Computación y procesamiento numérico

*El sistema UNIX dispone de muchas herramientas para computación. Desde la más simple (**expr**) hasta herramientas mucho más sofisticadas. En este capítulo nos centraremos en estas herramientas de que dispone el sistema estándar, y que pueden ser usadas sobre un terminal ASCII remoto además de sobre la consola del sistema.*

13.1.- LA ORDEN **EXPR**

El comando **expr** sirve evaluar expresiones aritméticas. El resultado de la evaluación se escribe en la salida estándar.

Sintaxis

expr expresión

La expresión está compuesta por varios literales (números enteros o cadenas de caracteres) y operadores, separados siempre por espacios. Los operadores que coincidan con caracteres especiales para el shell (p.e. los paréntesis) deberán ser precedidos por una barra inversa o encerrados entre comillas para que pierdan su significado especial. La tabla siguiente contiene los operadores más importantes admitidos por el comando **expr**. Existen operadores aritméticos, lógicos y de manejo de cadenas. Los operadores de comparación devuelven 0 como falso y 1 como verdadero; los operadores lógicos utilizan el valor 0 como falso y cualquier número distinto de 0 como verdadero. Los operadores no se pueden utilizar como literales de tipo cadena.

Operador	Significado
+	Suma de enteros.

Operador	Significado
-	Resta de enteros.
*	Producto de enteros.
/	División de enteros.
%	Módulo (resto de la división) entre enteros.
<	Comparador <i>menor que</i> .
<=	Comparador <i>menor o igual</i> .
=	Comparador <i>igual a</i> .
==	Comparador <i>igual a</i> (idéntico a =).
!=	Comparador <i>distinto de</i> .
>=	Comparador <i>mayor o igual</i> .
>	Comparador <i>mayor</i> .
	OR lógico.
&	AND lógico.
()	Permiten especificar el orden de las operaciones.
substr	Devuelve una subcadena. El formato es <i>substr cadena posición longitud</i> , y devuelve los <i>longitud</i> caracteres a partir de la posición <i>posición</i> de la cadena <i>cadena</i> .
length	Devuelve la longitud de una cadena. El formato es <i>length cadena</i> .
index	Devuelve la posición de un carácter en una cadena. El formato es <i>index cadena muestra</i> , y devuelve la posición en la cadena <i>cadena</i> de la primera aparición del primer carácter de la cadena <i>muestra</i> .

El orden de precedencia es el siguiente: operador *OR*, operador *AND*, operadores de comparación, operadores de adición, operadores de multiplicación.

El siguiente ejemplo de uso de este comando sirve para sumar 1 a la variable `CONTADOR`:

```
|  CONTADOR=`expr $CONTADOR + 1` |
```

Generalmente se definen variables del entorno para almacenar números, y, posteriormente, podemos usar su valor en otras órdenes **expr** o en operaciones **test**.

13.2.- LAS CALCULADORAS *DC* Y *BC*

Dos potentes *calculadoras* orientadas a línea vienen con el sistema estándar: las herramientas **dc** y **bc**. La calculadora **bc** es realmente un preprocesador para **dc**, pero ambas herramientas implementan los dos modelos diferentes de interfaz de usuario de las calculadoras modernas. La calculadora **dc** usa notación *postfija*, la llamada notación *polaca inversa*, en la que se introducen los cálculos tecleando dos números y luego el operador.

\$ dc

2

3

+

p

5

Aquí introduciremos **2** y **3**, luego el operador aritmético + y luego **p** para visualizar el resultado. El calculador **dc** imprime la respuesta: **5**. La orden **bc**, por su parte, usa *notación infija*, que es similar a las operaciones aritméticas normales.

\$ bc

2 + 3

5

Aquí introducimos **2+3** y **bc** imprime el resultado. Podemos usar cualquiera de las dos órdenes, pero las características de ambas calculadoras difieren ligeramente, y una puede ser más conveniente que la otra para ciertas operaciones. Ambas permiten cualquier *precisión* y pueden calcular en cualquier base o *raíz* numérica además de en la decimal normal.

13.2.1.- La orden *dc*

Después de ejecutarse la orden **dc**, ésta lee instrucciones en su lenguaje interno desde la entrada estándar. Alternativamente, podemos nombrar un solo fichero sobre la línea de orden, en cuyo caso se lee ese fichero hasta que finaliza, y luego **dc** pasa a leer su entrada estándar solicitando más órdenes. La entrada proveniente de un fichero es tratada de forma idéntica a las instrucciones provenientes de la entrada estándar, excepto que una instrucción fin-de-fichero [ctrl.+d] desde el teclado no acaba la ejecución de **dc**, mientras que el final del fichero de entrada no. La orden **q** [quit] también acaba **dc** desde cualquier fuente de entrada.

Una instrucción para **dc** puede ser un número, que es introducido inmediatamente en la pila **dc**. Los números pueden contener puntos decimales, y los números negativos están precedidos no por – sino por el carácter _ (subrayado). Por ejemplo, las siguientes órdenes son números aceptables para **dc**.

123

123.5

123.456789012345

_23.4

La calculadora **dc** reconoce los operadores aritméticos + (suma), - (resta), / (división), * (producto), % (resto) y ^ (exponenciación), que producen la acción apropiada sobre los dos números superiores de la pila. El resultado de la operación se coloca en la pila, reemplazando a esos dos números.

Un gran número de órdenes especiales nos permiten operaciones de control adicionales. La orden **p**, por ejemplo, hace que el valor superior de la pila sea visualizado pero no modifica la pila.

```
$ dc
3
4
*
p
12
q
$
```

En estos ejemplos cada orden está mostrada sobre una línea de entrada separada, pero **dc** puede aceptar varias órdenes en una sola línea si es necesario. Los números están delimitados por espacios en blanco, pero otras órdenes pueden ir juntas. La operación anterior podría también haberse escrito como sigue:

```
$ dc
3 4 * p q
$
```

Además, **dc** puede efectuar varias operaciones de golpe cuando hay varios números en la pila. Ya que el resultado de una operación reemplaza en la pila a los números usados en la operación, la siguiente operación usará ese valor y el número siguiente por debajo en la pila. Por ejemplo,

```
$ dc
3
4
7
+ - p
-8
```

Aquí el operador + suma los dos números superiores, 7 y 4, produciendo 11, que es luego colocado en la pila. Después el operador – resta 11 de 3, produciendo -8.

Por omisión, los cálculos retienen tantos dígitos decimales como sean necesarios. Todos los cálculos se efectúan sobre enteros a menos que se modifique el *factor de escala* con la orden **k**. Los operadores de suma y resta trabajan con todos los dígitos incluidos en los números de entrada; sólo las operaciones de multiplicación y división devuelven resultados que dependen del factor de escala.

Otras órdenes orientadas a pila son **c** para borrar, **d**, para duplicar el elemento superior de la pila, y **f** para imprimir todos los elementos de la pila en orden.

13.2.1.1.- Variables en *dc*

No se dispone de una operación para extraer simplemente el elemento superior de la pila y descartarlo. Sin embargo, **dc** permite variables *registro*. Los elementos de la pila pueden ser llevados a los registros, y las variables registro pueden ser almacenadas en la pila. Los nombres de los registros son caracteres minúsculas simples, por lo que en cualquier momento se permiten 26 variables registro. La orden **s** toma un nombre de registro a continuación y lleva la cima de la pila al registro nombrado. La orden **l** es la inversa. Extrae el valor del registro nominado y lo introduce en la pila.

2.34

4.56

p

4.56

st

p

2.34

lt

p

4.56

Para descartar un valor de la cima de la pila, usaremos **s** para colocarlo en un registro sin uso. Este procedimiento de empleo de un registro intermedio también sirve para *invertir* el orden de los elementos de la pila.

Las órdenes **s** y **l** pueden trabajar también con pilas auxiliares. Si el nombre de registro es un carácter mayúscula, entonces es tratado como una pila. Si la orden es **s**, el valor superior de la pila principal se introduce en la pila auxiliar.

2.34

```
sT
```

```
p
```

```
empty snack
```

```
5.44
```

```
sT
```

```
p
```

```
empty snack
```

```
lT
```

```
p
```

```
5.44
```

```
lT
```

```
p
```

```
2.34
```

Si una pila está vacía o no contiene números suficientes para completar una operación, se imprime el mensaje “empty snack”.

Una cadena de caracteres rodeada por [y] es tratada como una cadena ASCII en introducida en la pila.

```
[hola mundo]
```

```
p
```

```
hola mundo
```

Esta posibilidad de almacenar cadenas ASCII es una característica inusual para una calculadora. No podemos usar estas cadenas en operaciones numéricas, pero la orden **x** toma una cadena desde la cima de la pila y la ejecuta como orden **dc**. Debemos suprimir explícitamente la cadena de la pila después de ejecutar la orden, si esto es lo que deseamos. Además, la orden **!** es un escape al shell que hace que **dc** ejecute el resto de la línea en un subshell. Cuando la orden finaliza, **dc** reanuda su operación.

La calculadora **dc** permite muchas otras órdenes, incluyendo **v** para reemplazar el elemento superior de la pila por su raíz cuadrada y varias órdenes para cambiar la base o *raíz* numérica a usar en cálculos posteriores. La orden **i** hace que **dc** use el número que se halle en la cima de la pila como raíz para los números de entrada, mientras que **o** hace que **dc** use la raíz designada para salida. Esto permite conversión entre bases y cálculos en bases diferentes a la base 10.

```
2
```

```
i
1001

p
9
2
o
p
1001
```

Naturalmente, si se modifica la raíz de entrada, todos los números introducidos a partir de ese momento deben ser consistentes con la nueva raíz.

Si no estamos familiarizados con la notación postfija, el sistema **dc** puede resultar algo confuso.

13.2.2.- La calculadora *bc*

Aunque es realmente un preprocesador para **dc** y usa **dc** para hacer su trabajo, contiene muchas más funciones que **dc**, incluyendo funciones con nombre, operadores lógicos y funciones matemáticas tales como **sqr**t. El lenguaje de órdenes es globalmente parecido al lenguaje C, pero muco más simplificado. Al orden **bc** se ejecuta simplemente, y sus interacciones con **dc** no son visibles al usuario.

La calculadora **bc** comienza sigilosamente y espera nuestra entrada. Para salir de **bc** usaremos la orden **quit** o [ctrl.+d] para señalar el final de fichero. La calculadora **bc** lee su entrada estándar y escribe en su salida estándar, de modo que permite la redirección desde un fichero. Cuando el fichero finaliza, **bc** acaba.

```
$ cat fich.orden

6+5

[ctrl.+d]

$ bc <fich.orden

11

$
```

Además, **bc** puede tomar un nombre de fichero como argumento.

```
$ bc fich.orden
```

Esto es distinto a redirigir un fichero a la entrada estándar de **bc**. Cuando se especifica un nombre de fichero como argumento, **bc** lee el fichero y procesa sus órdenes y luego conmuta al terminal para la entrada. Esto nos permite almacenar

órdenes y funciones complejas en un fichero y luego posiblemente usarlas directamente desde el teclado.

13.2.2.1.- Notación *bc*

Se usa una notación *infija*, y el final de la línea de entrada señala que la orden debe ser evaluada.

Los números en **bc** pueden ser tan largos como sea necesario y pueden contener un punto decimal y un signo – (menos) opcional para indicar un número negativo. Los operadores aritméticos normales se permiten para los cálculos.

Se permiten variables, y éstas pueden tener nombres de un solo carácter en minúscula. Los valores se asignan a las variables con el operador =, como se muestra a continuación:

```
W=//valor//
```

La **w** puede ser cualquier letra minúscula. Las variables se usan como almacén temporal de números para uso posterior y pueden ser referenciadas como números.

```
$ bc
```

```
y=4
```

```
3+y
```

```
7
```

Las variables retienen sus valores hasta que sean reutilizadas en otra sentencia de asignación. Además, **bc** acepta vectores de números si los operadores **[y]** rodean al índice del vector.

```
S[2]=3.3
```

Los índices del vector comienzan en 0, de modo que este ejemplo designa al tercer elemento del vector de nombre **s** y le asigna el valor 3.3. El índice de un vector puede ser cualquier expresión que **bc** pueda resolver a un número.

Por omisión, **bc** efectúa muchos cálculos como si los números fueran enteros, pero podemos cambiar el factor de escala mediante la asignación de un número a la variable *scale*. El número es el número de dígitos a la derecha del punto decimal que deseamos que **bc** retenga en sus cálculos.

```
$ bc
```

```
6.456/5.678
```

```
1
```

```
scale=3
```

```
6.456/5.678
```

1.137

Además, podemos usar las variables *ibase* y *obase* como en **dc** para definir la raíz numérica de entrada y la raíz numérica de salida, respectivamente.

```
$ bc
ibase=2
1001
9
obase=8
1001
11
```

Podemos usar estas facultades para efectuar conversiones de base o para efectuar aritmética en sistemas numéricos diferentes al decimal.

La calculadora **bc** permite muchos otros operadores además de los operadores matemáticos normales entendidos por **dc**. Los operadores **++** y **--** incrementan y decrementan el valor de una variable, respectivamente.

También modifican el valor de una variable y devuelven el valor establecido, de modo que esta operación funcionaría así:

```
S=4
T[S--]=3.3
T[3]
3.3
```

Aquí, **t[4]** está indefinido, y **s** es 3 después de la operación. Los operadores **++** y **--** pueden ser usados antes o después del nombre de la variable. Cuando se usan antes del nombre, el valor se modifica antes de ser devuelto; cuando se usan después del nombre, el valor se modifica después de que el valor sea devuelto.

```
S=4
T[S--]=3.3
T[4]
3.3
```

En ambos casos el valor final de **s** es 3, pero en vector **t** difiere.

13.2.3.- La orden *awk*

Esta herramienta es muy útil tanto para cálculos como para tareas de *procesamiento de patrones*. El nombre **awk** es un acrónimo formado por el nombre de

sus tres creadores. Es realmente un lenguaje de programación muy potente y elegante que nunca ha logrado superar la rémora de una documentación extremadamente concisa y arcana. La orden **awk** puede hacer cosas que ninguna otra herramienta puede lograr sin extremas contorsiones. Aunque bastante ineficiente en operación, es extremadamente útil.

La orden **awk** explora una lista de ficheros de entrada buscando líneas que se correspondan con un conjunto de patrones especificados. Por cada patrón que coincida, se efectúa un conjunto especificado de acciones. Estas acciones pueden implicar manipulaciones de campos dentro de la línea u operaciones aritméticas sobre los valores de los campos. La herramienta **awk** es un lenguaje de programación con características de los lenguajes de programación shell, **bc** y C. Es completamente interpretado, como **bc**, contiene variables de campo en cada línea de entrada denominadas como los argumentos del shell **\$1**, **\$2** y **\$3**, y contiene operadores de impresión y control similares a los del lenguaje C.

Para usar **awk**, debemos crear un programa que especifique una lista de secciones patrón y acción. La orden **awk** lee los ficheros de entrada y, por cada línea de entrada que corresponda a un patrón, ejecuta la acción asociada. **Awk** se ejecuta con esta línea de orden.

```
$ awk -f prog lista-de-ficheros
```

En el formato **awk**, el conjunto de patrones y acciones está especificado en un fichero nombrado tras la opción **-f**. Cualesquiera nombres de ficheros adicionales son los ficheros de texto que **awk** lee para efectuar sus acciones. Si no se especifica una lista de ficheros, **awk** lee su entrada estándar.

También podemos insertar entrada estándar en mitad de una lista de ficheros con el argumento especial **-** (menos). Por ejemplo,

```
$ awk -f prog fichero1 - fichero2
```

Aquí **awk** leerá el programa desde el fichero **prog**, procesará **fichero1**, leerá su entrada estándar hasta que alcance una marca de fin-de-fichero y luego procesará **fichero2**.

La orden **awk** también permite que el programa sea incluido directamente en la línea de orden, si no se usa la opción **-f**. En este caso el programa aparece literalmente a continuación del nombre **awk** pero antes de la lista de ficheros. Sin embargo, incluso los expertos en **awk** necesitarán varias pruebas antes de depurar un programa **awk**, de modo que la forma en línea es raramente usada excepto en guiones shell cuidadosamente preparados. El uso en línea puede ahorrar un fichero adicional en una aplicación, pero debemos asegurarnos de que el programa funcione correctamente antes de prescindir de la forma **-f fichero**.

12.2.3.1.- Cómo lee *awk* las líneas de entrada

Cada línea leída desde los ficheros o desde la entrada estándar es tratada por **awk** como compuesta por campos separados por espacios en blanco. Podemos cambiar el delimitador de campo por cualquier otro carácter si

especificamos la opción **-F** en la línea de orden **awk**, con un nuevo delimitador como argumento. Por ejemplo, para hacer que **awk** use (:) como delimitador, podríamos usar la siguiente línea de orden,

```
$ awk -F: -f prog ficheros
```

Podemos referirnos a cada campo en la línea de entrada con los nombres especiales **\$1**, **\$2**, **\$3** y así sucesivamente. El primer campo de la línea es **\$1**. La variable especial **\$0** se refiere a la línea de entrada como un todo, sin estar dividida en campos.

12.2.3.2.- Patrones y acciones de *awk*

Las parejas patrón-acción definen las operaciones que **awk** efectúa sobre las líneas y campos que lee. El formato de estas parejas patrón-acción es como sigue,

```
Patrón {acción}
```

Se separa la porción *acción* de la porción *patrón* encerrándola entre llaves. La falta de acción hace que la línea sea impresa. La falta de patrón selecciona siempre la línea; es decir, si el patrón falta, la acción se aplica a todas las líneas. La acción puede contener una serie compleja de operadores, incluyendo variables y operadores lógicos, de forma parecida a **bc**, con las variables de campo **\$1** y **\$2**, etc., usadas como datos de entrada.

Un operador habitual es **print**, que escribe sus argumentos en la salida estándar. La acción

```
{ print $2, $1 }
```

invertirá los dos primeros campos de entrada y los escribirá. Si los datos de entrada son

```
$ cat fich.ent
```

```
hola adiós otra vez
```

```
111 222
```

```
treinta cuarenta
```

```
$
```

y el programa **awk** es

```
$ cat awk.prog1
```

```
{ print $2, $1 }
```

```
$
```

la salida será

```
$ awk -f awk.prog1 fich.ent
```

```
adiós hola

222 111

cuarenta treinta

$
```

Los argumentos para **print** en este ejemplo están separados por comas, lo que hace que **print** inserte el delimitador de campo actual entre los datos de salida. Si faltara la coma **\$1** y **\$2** irían juntos en la salida.

Debemos recordar que cuando el patrón se corresponde con la línea de entrada, la acción se lleva a cabo. En el ejemplo anterior no había sección patrón, de modo que la acción fue realizada sobre todas las líneas de entrada.

Un patrón es una expresión regular o secuencia de expresiones regulares separadas por los operadores **!** (no), **|** (or), **&&** (and) o paréntesis para su agrupación. Debemos incluir cada expresión regular entre **/** como se hace con **ed**. Por ejemplo, el programa **awk**

```
/hola/ { print $2, $1 }
```

procesará el **fich.ent** especificado arriba.

```
$ awk -f awk.prog2 fich.ent
```

```
adiós hola

$
```

Sólo una línea del fichero de entrada coincidía con el patrón **/hola/**, por lo que sólo se imprimió una línea con la parte de acción. Ya que no se especificaba acción para otras líneas, no se llevó a cabo ninguna y nada apareció en la salida. Sin embargo, se permiten múltiples sentencias patrón-acción para manejar diferentes casos.

```
/hola/ { print $2, $1 }

/treinta/ { print $1, $2, "y más2" }
```

Este programa da resultados diferentes.

```
$ awk -f awk.prog3 fich.ent
```

```
adiós hola

treinta cuarenta y más

$
```

El programa **awk** entero, incluyendo todas las parejas diferentes patrón-acción, será procesado para cada línea de entrada. Este ejemplo ilustra otra característica de

la orden **print**: entre los argumentos de **print** pueden aparecer cadenas literales rodeadas por marcas de acotación, y las cadenas aparecerán en la salida.

Podemos unir los patrones con operadores lógicos para ampliar el rango de tests posibles. Por ejemplo, el programa

```
/hola/||111/ { print "di", $1, $2 }
```

producirá

```
$ awk -f awk.prog4 fich.ent
```

```
di hola adiós
```

```
di 111 222
```

```
$
```

El operador **||** dice a **awk** que efectúe la acción si alguna de las dos expresiones regulares se corresponden con la línea de entrada, el operador **&&** dice a **awk** que efectúe la acción sólo si ambas expresiones regulares se corresponden con la línea de entrada, y el operador **!** dice a **awk** que efectúe la acción solamente si la expresión regular no se corresponde con la línea de entrada. El operador **!** precede a una expresión regular y no separa dos expresiones.

```
!/hola/ {  
    print "no hola"  
}
```

Naturalmente se permiten expresiones regulares más complejas. Esta orden

```
/^[Hh1]/ { print "di", $0 }
```

producirá

```
$ awk -f awk.prog5 fich.ent
```

```
di hola adiós otra vez
```

```
di 111 222
```

```
$
```

Observemos el uso de **\$0** para visualizar la línea de entrada original íntegra.

La parte de acción puede ocupar muchas líneas tecleadas si es necesario, cada sentencia adicional de la acción en una línea separada.

La acción completa va encerrada entre llaves. Por ejemplo:

```
$ cat >awk.prog6
```

```

/hola/ {

    print $2

    print "otro"

    print $1

}

$ awk -f awk.prog6 fich.ent

adiós

otro

hola

$

```

Cada sentencia **print** produce una nueva línea de salida, pero toda la salida está producida por la única línea del fichero de entrada que se corresponde con el patrón **/hola/**.

12.2.3.3.- Operaciones numéricas con **awk**

Como estos ejemplos simples muestran, **awk** dispone de excelentes facilidades para comparar y manipular cadenas. De hecho un uso principal de **awk** es reformatear datos de texto según reglas especificadas en su programa, ya que **awk** permite operaciones de reformateado que son demasiado complejas para **sed** u otras herramientas. Sin embargo, la potencia real de **awk** se halla en los operadores lógicos y aritméticos que se pueden emplear en la parte de acción. La orden **awk** es muy inteligente en el uso de variables numéricas y dispone de herramientas para conversión entre cadenas de caracteres y números. El uso de **awk** para la aritmética difiere del uso de **bc** en que **awk** puede usar la sección patrón para escoger un subconjunto de las líneas en un fichero de entrada. Esto permite que algunos campos contengan información clave y otros campos contengan los datos a ser procesados por **awk**. Además, **awk** puede convertir fácilmente entrada entre formatos de caracteres y numéricos, e incluye mejores herramientas para formatear la salida que **bc**. Por otra parte, **bc** es más eficiente, tiene mayor precisión y, de alguna manera, es más fácil de aprender.

Una de las características más amistosas de **awk** es que interpreta automáticamente los campos de entrada como cadenas de caracteres o como números según sea apropiado para el contexto. Por ejemplo, la función **awk** predefinida **length** devuelve la longitud de un campo de entrada interpretado como cadenas de caracteres, mientras que las variables numéricas pueden acoger el valor del campo interpretado como un número. Por ejemplo,

```

$ cat >awk.prog7

{

    s+= $2

```

```

        print $2, "longitud=" longitud($2), "s=" s
    }
$

```

Este ejemplo suma los valores numéricos del segundo campo de cada línea de entrada. La conversión de tipo es efectuada automáticamente por **awk**, y nosotros no tenemos que declarar la variable **s** por adelantado ni preocuparnos acerca de su tipo. Cuando este programa se ejecute, éstos serán los resultados.

```

$ awk -f awk.prog7 fich.ent

adiós longitud=5 s=0

222 longitud=3 s=222

cuarenta longitud=8 s=222

{

```

Las cadenas que no pueden ser convertidas a números toman el valor cero de modo que las cosas funcionan como es de esperar. La cadena **cuarenta** es de esta forma y no puede ser convertida, pero la cadena **222** puede ser convertida correctamente por **awk**. Por omisión, una forma tal como **222** es tratada como un número, aunque podemos forzar a **awk** a tratarla como una cadena si la entrecorrimos.

La orden **awk** nos permite asignar valores a variables, como en **s=0**. Las variables en **awk** son tratadas de forma parecida a las variables en **bc**, excepto que los nombres de las variables en **awk** pueden ser de más de un carácter. De hecho, pueden ser de cualquier longitud siempre que comiencen por un carácter alfabético. Se permiten variables vectores, con el índice del vector encerrado entre corchetes como en **bc**. Todas estas son variables **awk** legales.

```

s

S

SS

S1

Qwerty[42]

```

No necesitamos declarar o inicializar las variables antes de usarlas. La orden **awk** inicializa las variables a la cadena vacía, pero podemos usarlas para almacenar una cadena o un número según deseemos, sin dificultad. De hecho, una sola variable puede modificar su tipo mientras es usada en una acción. Por ejemplo,

```

/hola/ {

    SSS=34

    print "SSS es", SSS
}

```

```
    SSS= hola

    print "SSS es", SSS

}
```

podría producir la salida siguiente:

```
$ awk -f awk.prog8 fich.ent

SSS es 34

SSS es hola
```

Esta tipificación automática hace que las variables de **awk** sean fáciles de usar, y **awk** protestará si tratamos de usar esta potencia inadecuadamente

Las funciones de **awk** se extienden a través de un territorio tan extenso como complejo. Sin embargo, con el esbozo anterior creemos haber dado una idea de cómo usar esta herramienta que incorpora UNIX. Sin embargo, el uso de funciones con **awk** sería tema de otro enorme manual como éste que ahora estamos escribiendo. Por ello remitimos a manuales especializados en **awk** para comprender más acerca de las funciones y desarrollo en **awk**.

Editores de texto

El sistema UNIX suministra varios editores de texto diferentes de forma estándar. Estos editores se diferencian grandemente y están generalmente optimizados para un subconjunto del trabajo de edición de texto. Ninguno de ellos es un verdadero procesador de textos. Las herramientas de edición de texto están diseñadas como una interfaz de usuario muy rápida y concisa y con una forma de operación que favorece al experto experimentado a expensas del principiante. Generalmente, si algo no nos sale en el editor es porque no sabemos hacerlo, y no porque el editor sea incapaz.

En Linux se incluyen multitud de editores de texto. Por ejemplo,

- *joe*: se caracteriza por que utiliza las mismas combinaciones de teclas que los editores *Micro-Pro WordStar* o los editores de los IDE de Borland "*Turbo*"
- *emacs*: un editor muy grande, completo y extendido. Además puede funcionar bajo entorno gráfico.
- *vi*: bastante complicado de manejar, pero es importante conocerlo por que es el estándar en UNIX (lo encontraremos en cualquier máquina UNIX).
- *wpe*: un entorno de programación consistente en un editor similar a los editores de los IDE de Borland "*Turbo*" más modernos, que integra la invocación de distintos compiladores y un entorno de traza.
- *xwpe*: la versión gráfica bajo X-Window de *wpe*.

14.1.- EL EDITOR *JOE*

El editor *joe* se caracteriza por que utiliza las mismas combinaciones de teclas que los editores *Micro-Pro WordStar* o los editores de los IDE de Borland "*Turbo*", por lo que nos puede resultar muy familiar.

Al ejecutar el editor, admite como parámetro el nombre del fichero que queremos editar. No estudiaremos las opciones. No obstante, podemos conocerlas con ayuda del manual (`man joe`).

joe [opciones] [fichero]

La mayoría de las combinaciones de teclas que utiliza *joe* están formadas por dos pulsaciones consecutivas. Por ejemplo, con [^]KH (pulsamos la tecla *Ctrl* y la tecla *K*, y luego la tecla *H*) *joe* nos muestra una pantalla de ayuda con las combinaciones de teclas más importantes. Otras combinaciones se hacen en una sola pulsación. Por ejemplo, con [^]Y (pulsamos la tecla *Ctrl* y la tecla *Y*) borramos una línea del fichero.

La tala adjunta contiene las principales combinaciones de teclas que se utilizan en el editor *joe*.

Combinación	Categoría	Significado
[^] KD	Archivos	Guarda el fichero actual
[^] KE	Archivos	Carga un fichero de disco para editarlo.
[^] KB	Bloques	Marca el inicio de un bloque.
[^] KC	Bloques	Copia el bloque marcado a la posición actual.
[^] KK	Bloques	Marca el final de un bloque.
[^] KM	Bloques	Mueve el bloque marcado a la posición actual.
[^] KR	Bloques	Inserta el contenido de un archivo en la posición del cursor.
[^] KW	Bloques	Guarda el bloque marcado en un fichero.
[^] KY	Bloques	Borrar el bloque marcado.
[^] KF	Búsqueda	Permite especificar un texto para ser buscado en el fichero.
[^] L	Búsqueda	Repite la última búsqueda realizada (busca la siguiente aparición).
[^] [^]	Edición	Vuelve a hacer los últimos cambios deshechos (en los teclados en español, esta tecla es [^] ' (<i>Ctrl</i> más la tecla de apóstrofe). En conexiones vía <i>telnet</i> posiblemente esta combinación no funcione.
[^] _	Edición	Deshace los últimos cambios. En conexiones vía <i>telnet</i> posiblemente esta combinación no funcione.
[^] D	Edición	Borra el carácter situado a la derecha del cursor.
[^] O	Edición	Borra la palabra situada a la izquierda del cursor.
[^] W	Edición	Borra la palabra situada a la derecha del cursor.
[^] Y	Edición	Borra la línea en la posición del cursor.
Del	Edición	Borra el carácter situado a la izquierda del cursor.
[^] A	Movimiento	Mueve el cursor al principio de la línea.
[^] E	Movimiento	Mueve el cursor al final de la línea.
[^] KL	Movimiento	Mueve el cursor a un número de línea que se introduce por teclado.

Combinación	Categoría	Significado
^KU	Movimiento	Mueve el cursor al principio del fichero.
^KV	Movimiento	Mueve el cursor al final del fichero.
^U	Movimiento	Retrocede una página.
^V	Movimiento	Avanza una página.
^X	Movimiento	Mueve el cursor una palabra a la derecha.
^Z	Movimiento	Mueve el cursor una palabra a la izquierda.
^C	Varios	Termina y sale al <i>shell</i> .
^KH	Varios	Muestra u oculta la ventana de ayuda sobre el teclado.
^KX	Varios	Guarda el fichero actual, termina y sale al <i>shell</i> .
^KZ	Varios	Sale al <i>shell</i> para ejecutar comandos (se puede volver con <i>exit</i>).
^T	Varios	Accede al menú de configuración de modo de edición, que permite cambiar modo de inserción, modo de indentación, modo de corte de palabras, tamaño de las tabulaciones, márgenes, etc.

Podemos crear un fichero `~/.joerc` con las opciones predefinidas del editor, entre ellas, el contenido de la ventana de ayuda y la disposición del teclado (combinaciones de teclas). Podemos utilizar como muestra el archivo `/usr/lib/joe/joerc`.

14.2.- EL EDITOR *EMACS*

El editor *emacs* es parecido a *joe*, pero mucho más completo y complejo. Estudiaremos únicamente su funcionamiento como editor básico de textos, pero en realidad su utilidad es mucho más amplia. La instalación completa puede ocupar mucho espacio en disco, por lo que suelen existir opciones especiales en la instalación de Linux para incluir o no los componentes de *emacs*.

Al ejecutar el editor, admite como parámetro el nombre del fichero que queremos editar.

<i>Sintaxis</i>
emacs [<i>fichero</i>]

Algunas de las combinaciones de teclas que utiliza *emacs* están formadas por dos pulsaciones consecutivas. Por ejemplo, con *^HH* (pulsamos la tecla *Ctrl* y la tecla *H*, y luego la tecla *H*) *emacs* nos muestra una pantalla de ayuda, de la que podemos salir con *^XL*. Otras combinaciones se hacen en una sola pulsación. Por ejemplo, con *^A* (pulsamos la tecla *Ctrl* y la tecla *A*) saltamos al principio de una línea. También hay combinaciones que utilizan dos o más veces la tecla *Ctrl*. Finalmente, algunas combinaciones comienzan con la tecla *Escape* seguida de una pulsación. Por ejemplo, *Escape F* (pulsamos la tecla *Escape* y luego la tecla *F*)

Combinación	Categoría	Significado
^X^F	Archivos	Carga un fichero en otra ventana.
^X^R	Archivos	Carga un fichero, eliminando el contenido actual.
^X^S	Archivos	Guarda el fichero actual.
^X^W	Archivos	Guarda el fichero actual con otro nombre.
^XN	Archivos	Guarda el fichero actual con otro nombre.
^Espacio	Bloques	Marca el principio de un bloque ⁴ .
^W	Bloques	Elimina el bloque marcado.
^X^I	Bloques	Inserta un archivo en la posición del cursor.
^Y	Bloques	Pega el último bloque cortado (borrado con algún comando) o copiado al portapapeles.
Escape w	Bloques	Copia el bloque marcado al portapapeles.
^R	Búsqueda	Busca una cadena hacia atrás.
^S	Búsqueda	Busca una cadena.
^XR	Búsqueda	Repite la búsqueda hacia atrás.
^XS	Búsqueda	Repite la búsqueda.
Escape ^R	Búsqueda	Busca y reemplaza de forma interactiva.
Escape r	Búsqueda	Busca y reemplaza (sin preguntar).
^D	Edición	Borra un carácter a la derecha.
^K	Edición	Borra hasta el final de la línea.
^T	Edición	Intercambia el carácter actual con el que le sigue.
^X^T	Edición	Intercambia la línea actual con la que le sigue.
Escape c	Edición	Pone en mayúsculas los caracteres necesarios.
Escape d	Edición	Borra una palabra a la derecha.
Escape l	Edición	Pone una palabra en minúsculas (todos los caracteres).
Escape t	Edición	Intercambia la palabra sobre la que está el cursor con la que le sigue.
Escape u	Edición	Pone una palabra en mayúsculas (todos los caracteres).
^A	Movimiento	Salta al principio de una línea.
^B	Movimiento	Mueve el cursor a la izquierda (←).
^E	Movimiento	Salta al final de una línea.
^F	Movimiento	Mueve el cursor a la derecha (→).
^N	Movimiento	Mueve el cursor abajo (↓).
^P	Movimiento	Mueve el cursor arriba (↑).
^V	Movimiento	Avance de página.

⁴ Trabajar con bloques, marcamos el principio del bloque con **^Espacio**, a continuación movemos el cursor para marcar el final, y por último efectuamos alguna operación con el bloque, por ejemplo, con **^W** borramos el bloque.

Combinación	Categoría	Significado
<code>^X]</code>	Movimiento	Avance de página.
<code>Escape :</code>	Movimiento	Inicio del documento (primera línea).
<code>Escape ;</code>	Movimiento	Fin del documento (última línea).
<code>Escape a</code>	Movimiento	Salta a una línea.
<code>Escape b</code>	Movimiento	Mueve el cursor una palabra a la izquierda.
<code>Escape f</code>	Movimiento	Mueve el cursor una palabra a la derecha.
<code>Escape v</code>	Movimiento	Retroceso de página.
<code>^X^C</code>	Varios	Termina <i>emacs</i> .
<code>^X^U</code>	Varios	Deshace el último cambio.
<code>^XB</code>	Varios	Cambia a otra ventana (otro archivo en edición)
<code>^XX</code>	Varios	Cambia a la siguiente otra ventana (otro archivo en edición)
<code>Escape ^N</code>	Varios	Cambia el nombre de una ventana.
<code>Escape z</code>	Varios	Guarda todos los ficheros y termina.

En los comandos de búsqueda, mientras se teclea la palabra buscada, *emacs* realiza la búsqueda (esto se llama *búsqueda incremental*). La búsqueda con reemplazamiento interactiva nos permite decidir qué hacer en cada aparición del patrón buscado. Entonces podemos responder con las opciones de la tabla siguiente.

Opción	Acción
<code>^G</code>	Cancelar.
<code>?</code>	Obtener una lista de opciones.
<code>.</code>	Sustituir y terminar, devolviendo el cursor a la posición inicial.
<code>,</code>	Sustituir el resto del documento (pasa a modo no interactivo).
<code>y</code>	Sustituir y buscar la siguiente aparición.
<i>Espacio</i>	Sustituir y buscar la siguiente aparición (igual que <code>y</code>).
<code>n</code>	No sustituir y buscar la siguiente aparición.

Se puede repetir varias veces de forma automática un comando pulsando *Escape* y un número, y a continuación ejecutando el comando.

Podemos crear un archivo llamado `.emacs` en nuestro directorio *home* para definir las opciones del editor.

emacs puede funcionar con un entorno de menús. Para ello es necesario trabajar bajo un entorno gráfico. Por esta razón, no podemos utilizar esta facilidad a través de una conexión de tipo *telnet*. En cambio, podemos emplear sistemas de conexión mediante gráficos como *eXcursion*, en lugar de *telnet* o cualquier tipo de conexión basada en texto.

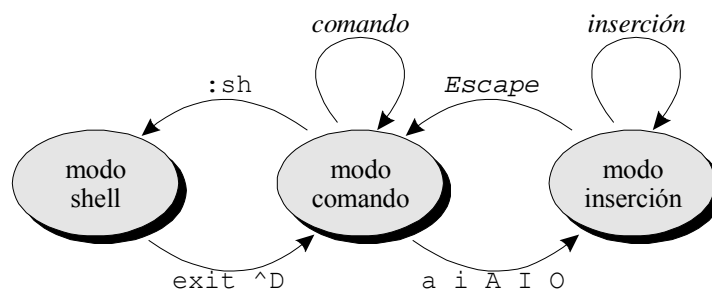
14.3.- EL EDITOR VI

vi es el editor de pantalla más extendido en UNIX. Es bastante difícil de manejar, pero conviene conocerlo puesto que es el que nos podemos encontrar en cualquier máquina UNIX. En los sistemas Linux, *vi* conserva el funcionamiento original, pero se le han añadido nuevas combinaciones de teclas para hacerlo más intuitivo, al menos, a los usuarios de MS-DOS/Windows. Nosotros estudiaremos el funcionamiento original. El editor se invoca pasando como parámetro, opcionalmente, el nombre del fichero que queremos editar.

Sintaxis

vi [*fichero*]

vi funciona con 3 modos: modo *shell*, modo *comando* y modo *inserción*. Podemos pasar de un modo a otro en cualquier momento pulsando las combinaciones de teclas o comandos adecuados. La **¡Error! No se encuentra el origen de la referencia.** ilustra en forma de autómatas estos tres modos.



- El modo *shell* permite escribir y ejecutar comandos del shell. Una vez dentro de *vi*, podemos hacer una salida momentánea al shell, ejecutar algún comando, y regresar de nuevo a *vi* con *exit*.
- El modo *comando* nos permite trabajar con el texto: mover el cursor de edición, insertar líneas en blanco, borrar líneas, copiar líneas, borrar palabras, deshacer/rehacer, buscar textos, guardar, salir, etc. Los comandos se invocan mediante la pulsación de alguna tecla (distinguiendo entre mayúsculas y minúsculas) o bien mediante una palabra clave precedida por algún símbolo, por ejemplo, dos puntos (:).
- El modo *inserción* sirve para añadir texto al documento. Pulsando *Escape*, volvemos al modo de comandos.

La tabla siguiente contiene los principales comandos que nos permiten utilizar el editor *vi*.

Combinación	Categoría	Significado
:w	Archivos	Guarda el fichero en edición.

Combinación	Categoría	Significado
:w <i>fichero</i>	Archivos	Guarda el fichero en edición con el nombre <i>fichero</i> .
^G	Archivos	Muestra información sobre el fichero que estamos editando.
##Y	Bloques	Copia al portapapeles la línea actual y las ##-1 siguientes (para duplicarla con P o p).
:r <i>fichero</i>	Bloques	Inserta en la posición del cursor el contenido del fichero <i>fichero</i> .
p	Bloques	Inserta el último bloque borrado (dd) o copiado al portapapeles (Y) en la siguiente línea.
P	Bloques	Inserta el último bloque borrado (dd) o copiado al portapapeles (Y) en la línea actual.
##G	Búsqueda	Salta a la línea número ##, siendo ## cualquier número.
/patrón/	Búsqueda	Salta a la próxima ocurrencia del patrón.
:##	Búsqueda	Salta a la línea número ##.
?patrón?	Búsqueda	Salta a la anterior ocurrencia del patrón.
n	Búsqueda	Repite la última búsqueda.
N	Búsqueda	Repite la última búsqueda, pero en dirección inversa.
~	Edición	Cambia el carácter sobre el cursor de mayúsculas a minúsculas, o viceversa.
a	Edición	Pasa al modo de inserción, situando el cursor una posición a la derecha (<i>Escape</i> para terminar).
A	Edición	Pasa al modo de inserción, situando el cursor al final de la línea (<i>Escape</i> para terminar).
C	Edición	Elimina el resto de la línea a la derecha y pasa al modo de inserción (<i>Escape</i> para terminar).
cw	Edición	Elimina el resto de la palabra a la derecha y pasa al modo de inserción (<i>Escape</i> para terminar).
D	Edición	Borra hasta el final de la línea.
dd	Edición	Borra la línea actual (la pasa al portapapeles).
dw	Edición	Borra a la derecha del cursor hasta el final de palabra.
i	Edición	Pasa al modo de inserción, dejando el cursor en la posición actual (<i>Escape</i> para terminar).
I	Edición	Pasa al modo de inserción, situando el cursor al principio de la línea (<i>Escape</i> para terminar).
J	Edición	Une la línea actual con la siguiente.
o	Edición	Pasa al modo de inserción, situando el cursor en la línea siguiente (<i>Escape</i> para terminar).
rx	Edición	Reemplaza el carácter sobre el cursor por el carácter x.
x	Edición	Borra un carácter a la derecha (el situado sobre el cursor).
X	Edición	Borra un carácter a la izquierda (el situado antes del cursor).
:w	Ficheros	Guarda el fichero en edición.
-	Movimiento	Salta a la primera posición de la línea anterior.
\$	Movimiento	Salta al final de la línea actual.

Combinación	Categoría	Significado
^B	Movimiento	Retroceso de página.
^D	Movimiento	Avance de media página.
^E	Movimiento	Avance de una línea (↓)
^F	Movimiento	Avance de página.
^U	Movimiento	Retroceso de media página.
^Y	Movimiento	Retroceso de una línea (↑)
+	Movimiento	Salta a la primera posición de la línea siguiente.
0 (cero)	Movimiento	Mueve el cursor hasta el principio de la línea actual.
b	Movimiento	Retrocede el cursor a la palabra anterior.
B	Movimiento	Retrocede el cursor a la palabra anterior, contando los signos de puntuación como parte de la palabra.
Delete	Movimiento	Retroceso de un carácter (←)
e	Movimiento	Mueve el cursor hasta el final de la palabra actual.
E	Movimiento	Mueve el cursor hasta el final de la palabra, contando los signos de puntuación como parte de la palabra.
Enter	Movimiento	Salta a la primera posición de la línea siguiente.
Espacio	Movimiento	Avance de un carácter (→)
H	Movimiento	Saltar al inicio del documento.
L	Movimiento	Saltar al final del documento.
M	Movimiento	Situar el cursor en el centro de la pantalla.
w	Movimiento	Avanza el cursor a la palabra siguiente.
W	Movimiento	Avanza el cursor a la palabra siguiente, contando los signos de puntuación como parte de la palabra.
z	Movimiento	Centrar la pantalla de manera que la posición actual del cursor quede en la parte superior de la pantalla.
z.	Movimiento	Centrar la pantalla en la posición actual del cursor.
: !comando	Shell	Ejecuta el comando <i>comando</i> del shell.
: sh	Shell	Pasa a modo shell para que podamos ejecutar comandos (<i>exit</i> para regresar a <i>vi</i>).
: q	Varios	Abandona <i>vi</i> .
: q!	Varios	Abandona <i>vi</i> sin guardar los cambios.
: set	Varios	Muestra el estado de las opciones principales.
: set all	Varios	Muestra el estado de todas las opciones.
: wq	Varios	Guarda el fichero en edición y abandona <i>vi</i> .
: x	Varios	Guarda el fichero en edición y abandona <i>vi</i> .
u	Varios	Deshace los últimos cambios.
U	Varios	Rehace los últimos cambios deshechos.

Todas las operaciones de *vi* pueden ser antepuestas por un número, lo que hace que la operación se repita ese número de veces. Por ejemplo, `3dd` borra 3 líneas de una vez, `23P` pega el último texto borrado 23 veces, etc.

Los comandos de búsqueda utilizan el punto (.) como comodín para representar cualquier carácter. Para buscar un punto debe anteponerse el carácter "\". Por ejemplo, para buscar la cadena "etc." debemos escribir el siguiente comando: `"/etc\."`.

vi tiene varias opciones de configuración, que pueden ser consultadas con el comando `:set`. Este comando también permite establecer las opciones. Por ejemplo, para activar la opción de auto-indentación podemos poner `":set ai"` o `":set autoindent"`, y para desactivarla, `":set noai"` o `":set noautoindent"`. Otra opción importante que podemos activar es "nu" (o "number"), que muestra los números de línea a la izquierda de la pantalla. En el mismo comando `set` se pueden establecer varias opciones.

Las opciones también pueden establecerse en un archivo de configuración, en el directorio *home* de cada usuario. Este fichero debe llamarse `".exrc"` y debe contener una secuencia de ordenes para *vi*. Por ejemplo:

```
set autoindent autowrite report=2 showmatch
```

`".exrc"`

14.4.- ENTORNOS WPE Y XWPE

wpe (*window programming environment*) es un entorno de programación (no sólo un editor) semejante a los IDE de Borland (Borland Pascal, C, etc.). El aspecto (ver **¡Error! No se encuentra el origen de la referencia.**) es similar y tiene aproximadamente las mismas funciones incorporadas:

- **Facilidades de edición:** manejo de bloques, portapapeles, etc.
- **Editor orientado a sintaxis:** utilización de distintos colores para identificar las secciones de los programas.
- **Invocación al compilador:** sin necesidad de teclear desde línea de comando la llamada al compilador y sus parámetros.
- **Depurador integrado:** con las opciones habituales de traza: ejecución paso a paso, inspección de objetos (variables), puntos de ruptura, etc.

Por ahora sólo nos interesa su funcionamiento como editor. En general podemos utilizar las mismas teclas que en *joe* y que en los editores de Borland para MS-DOS, además de los menús.

Existe una versión para entornos gráficos (X-Window) llamada *xwpe*, idéntica a *wpe*, pero más vistosa. La figura siguiente muestra el aspecto de *xwpe*.

```

X Window Programming Environment
File Edit Search Block Run Debug Project Options Window Help
ejemplo.cace
while true
do
echo OPCIONES
echo -----
echo L - listar informaci n del fichero #1
echo V - visualizar contenido del fichero #1
echo E - editar fichero #1
echo R - borrar fichero #1
echo A - Abortar
echo
echo -n "Selecciona una opci n: "
read OPCION
clear
case $OPCION in
L) ls -l $1;;
E) vi $1;;
V) more $1;;
R) rm -f $1
break;;
A) break;;
*) echo Debes escribir la opci n en may sculas;;
*) echo Esa opci n no existe;;
esac
done

```

xvpe necesita trabajar bajo entorno gráfico, por lo que no podemos utilizarlo a través de una conexión de tipo *telnet*. Necesitamos algún sistema de conexión mediante gráficos como *eXursion*, o, por supuesto, trabajar en la consola principal bajo *X-window*

14.5.- EL EDITOR DE FLUJO *SED*

La orden **sed** es un filtro como **grep**, pero nos permite hacer modificaciones a los ficheros. **sed** lee su entrada línea a línea y escribe las líneas una a una en su salida estándar. Por cada línea leída, **sed** aplica una operación de sustitución de la forma utilizada con **ed**. Si se produce la coincidencia, se realiza la sustitución y se escribe la línea; si no hay coincidencia, la línea se escribe sin modificar.

Las líneas son leídas, modificadas y escritas una a una. Por tanto, no hay buffer en memoria del fichero completo, lo que significa que ficheros de cualquier tamaño pueden ser modificados con **sed**, incluso los que sean demasiado grandes para otros editores de texto. La orden **sed** se usa a menudo para editar ficheros que son mayores de un megabyte. La mayoría de los editores de texto normales no pueden manejar ficheros tan grandes.

La orden **sed** se invoca con una línea de orden como la de **grep**, excepto que se puede usar un operador de sustitución completo. Esta orden

```
$ sed "s/hola/adiós/" fich.ent
```

sustituirá la primera instancia **hola** en cada línea del fichero **fich.ent** con la cadena **adiós**, y escribirá la línea en la salida estándar.

```
$ echo "1234hola5678" | sed "s/hola7adiós/"
```

```
1234adiós5678
```


\$

Se acota la orden de sustitución para protegerla de interpretación por el shell. Como es habitual, la cadena de coincidencia puede ser cualquier expresión regular.

La orden **sed** también permite muchos otros operadores. Podemos suprimir todas las líneas que contengan la cadena **hola** con la orden siguiente:

```
$ sed "/hola/d" fich.ent
```

En este ejemplo la orden significa: “buscar la cadena **hola**, y, si se encuentra, suprimir la línea”. Esta orden **sed** tiene el mismo resultado que la siguiente:

```
$ grep -v hola fich.ent
```

Para suprimir únicamente la cadena **hola** de la línea, sin suprimir la línea entera de la salida, deberíamos usar esta forma en su lugar:

```
$ sed "s/hola/" fich.ent
```

Los operadores **sed** pueden tomar también una dirección de línea o un rango de direcciones si se desea restringir los cambios a una parte del fichero. Esta orden

```
$ sed "3,7s/hola/" fich.ent
```

suprimirá el primer **hola** de las líneas 3 a 7 del fichero y dejará el resto de las líneas inalteradas. Además, podemos usar una dirección de contexto en lugar de un número de línea, si así lo deseamos.

```
Sed "/hola/,/adiós/s/malo/bueno/g" fich.ent
```

Esta orden hallará la primera instancia de la cadena **hola**, cambiará todas las instancias de la cadena **malo** por **bueno** hasta que encuentre la cadena **adiós** o finalice el fichero. En este ejemplo, si hay otra instancia de **hola** después de hallarse **adiós**, entonces la sustitución comenzará de nuevo en la siguiente instancia de **adiós**.

Realmente **sed** es incluso más potente de lo que hemos descrito hasta ahora. Si colocamos la orden en un fichero en lugar de en la línea de orden, podemos usar **sed** con la opción **-f**.

```
$ sed -f fich.orden fich.ent
```

En esta orden los operadores de expresión regular están en el fichero **fich.orden**. Por lo demás, **sed** actúa como es de esperar. Con sólo una única orden, como en los ejemplos anteriores, el fichero de órdenes tiene poca utilidad aunque las expresiones regulares complejas pueden a veces ser más fáciles de depurar si se mantienen en un lugar relativamente permanente. Sin embargo, el fichero de órdenes tiene una función más importante: podemos escribir guiones multilíneas

para **sed** de modo que una serie de operaciones pueden ser efectuadas sobre cada línea de entrada antes de que **sed** la escriba en la salida. Por ejemplo, podemos crear un fichero de nombre **fich.orden** con la siguiente lista de órdenes:

```
s/hola/adiós/
```

```
s/diós/diablo/
```

Ahora, si ejecutamos

```
$ echo "1234hola5678" | sed -f fich.orden
```

la salida será

```
$ echo "1234hola5678" | sed -f fich.orden
```

```
1234adiablo5678
```

```
$
```

Las operaciones especificadas en el fichero se ejecutan secuencialmente sobre cada línea de la entrada hasta que se borre la línea o se alcance el fin del fichero. Cuando se completa el conjunto de órdenes, la línea se escribe en la salida estándar; luego se lee la siguiente línea de entrada, y el proceso se repite.

Otras órdenes

A lo largo de este manual hemos tratado y trataremos (en la parte de Administración), las órdenes más interesantes de UNIX. En este capítulo vamos a explicar algunas otras órdenes muy importantes pero que no tienen un lugar específico dentro del estudio de UNIX.

15.1.- LA ORDEN *EXPORT*

El comando `export` permite hacer que el valor de una variable del shell sea transmitido los programas que ejecute el shell. Admite como parámetro el conjunto de variables que queremos exportar.

Sintaxis

export [*variable*]

Por defecto, las variables definidas en un shell son locales a ese shell, y no se transmiten a los programas que ejecutemos desde él. Con este comando indicamos al shell la necesidad de que sí se transmitan a todos los programas que ejecutemos a partir de ese momento.

Si no se especifica ningún parámetro, `export` muestra una lista con las variables que se están exportando actualmente.

15.2.- LA ORDEN *SET*

El comando `set` permite ver el valor de todas las variables definidas. Además, permite activar y desactivar algunas opciones del shell.

Sintaxis

set

Por ejemplo:

```

Remigio:~$ set
BASH=/bin/bash
BASH_VERSION=1.14.7(1)
EUID=1000
HISTFILE=/home/Remigio/.bash_history
HISTFILESIZE=500
HISTSIZ=500
HOME=/home/Remigio
HOSTNAME=Remigio.diesia.uhu.es
HOSTTYPE=i386
HUSHLOGIN=FALSE
LOGNAME=Remigio
LS_OPTIONS=--8bit --color=tty -F -b -T 0
OSTYPE=Linux
PATH=.:usr/local/bin:/bin:/usr/bin:/usr/X11/bin:/usr/local/jdk/bin:/usr/
openwin/bin:/usr/local/pascalfc/bin:/usr/lib/teTeX/b
in:/usr/local/pgsql/bin
PGDATA=/usr/local/pgsql/data
PGLIB=/usr/local/pgsql/lib
PPID=27357
PS1=\h:\w\$
SHELL=/bin/bash
SHLVL=1
TERM=vt220
UID=1000
USER=Remigio

```

15.3.- LA ORDEN ALIAS

El comando `alias` permite ver el valor de las opciones de shell, así como activarlas y desactivarlas.

Sintaxis

alias [*nombre* = '*comando*[*opciones*]]

Para visualizar el conjunto de alias definidos, escribimos el comando sin opciones, y para definir un alias, escribimos como opción el nombre del alias seguido del símbolo = (sin dejar espacios en medio) y a continuación el comando equivalente con sus parámetros si fuera necesario, recomendablemente entre comillas para evitar sustituciones erróneas. Por ejemplo: `alias dir='ls -al'`.

Este comando admite las siguientes opciones:

Modificador	Resultado
-x	La definición de alias se exportará en scripts del shell que sólo se ejecutarán mediante el nombre del script.
-t	Permite el uso de los alias con seguimiento en los que se guarda el nombre de ruta completo del comando.

15.4.- LA ORDEN *WHO*

El comando *who* sirve para mostrar el listado de usuarios conectados en un momento dado a la máquina UNIX.

Sintaxis

who [*opciones*]

Si no se especifican opciones, *who* muestra un listado con la siguiente información, para cada usuario: Nombre de *login*, nombre del terminal desde el que se conecta, hora de entrada, dirección desde la que se conecta. Por ejemplo:

```
Remigio:~$ who
Remigio tty0    Oct 22 10:18 (jmal.diesia.uhu.)
ggaleano tty2   Oct 22 11:13 (ggg.diesia.uhu.)
root          tty1   Oct 22 11:03 (:0.0)
```

Las opciones que podemos tener aparecen reflejadas en la tabla siguiente:

Modificador	Resultado
-m	Who funciona como el comando <i>whoami</i> .
-T	Añade el campo estado a la lista de campos. Si fuera la única opción, el resto de los campos serían los indicados en la opción -i.
-s	Lista únicamente los campos nombre, línea y hora.
-q	Es la opción rápida de <i>who</i> ; sólo visualiza los nombres y el número de usuarios, ignorando las otras opciones

15.5.- LA ORDEN *WHO AM I* | *WHO I AM*

El comando *whoami* (*who am I?*) sirve para obtener nuestro nombre de usuario. No admite parámetros.

Sintaxis

who am i

```
Remigio:~$ whoami
Remigio
```

15.6.- LA ORDEN *TEST*

El comando `test` sirve para comprobar una condición. Como resultado, no ofrece ninguna salida. Si la condición comprobada es verdadera, termina con éxito (estado 0), y si no, termina con error (estado distinto de 0).

Sintaxis

test expresión

Este comando se utiliza con las estructuras `if` y `while` que permiten construir bucles iterativos en la programación de *scripts* para el shell . Para hacer más fácil su comprensión dentro de los de *scripts*, tiene una sintaxis alternativa. En este caso los corchetes no indican opcionalidad, sino que deben escribirse literalmente:

SINTAXIS ALTERNATIVA

[expresión]

En realidad el corchete izquierdo "[" es un programa, concretamente un enlace al comando `test`, situado en alguna parte del árbol de directorios. Por esta razón no debemos olvidar dejar un espacio de separación al menos entre el corchete izquierdo y el primer carácter de la expresión.

Las posibles expresiones se recogen en la tabla que se adjunta posteriormente. Todas las que se refieren a ficheros, son falsas si el fichero referido no existe. Además, se pueden combinar varias expresiones utilizando los operadores de la tabla que se expone más adelante. A continuación veremos algunos ejemplos de uso de `test`:

- Comprobar la existencia de un fichero llamado `/tmp/status`.
- Comprobar si el directorio actual es el raíz.
- Comprobar si el número de usuarios conectados es mayor que 5⁵.
- Comprobar si `/tmp/a35627` es un directorio o un fichero (no un dispositivo, ni un enlace, etc.).
- Comprobar que el directorio actual en el *home*, y que nuestro nombre de usuario no es *root*.

⁵ Utilizar el comando `who` para obtener un listado de usuarios conectados.

```
[ -f /tmp/status ]
[ "/" = `pwd` ]
[ `who | wc -l` -gt 5 ]
[ -d /tmp/a35627 -o -f /tmp/a35627 ]
[ \( $HOME = `pwd` \) -a \( ! $USER = root \) ]
```

Expresión	La expresión es verdadera cuando...
<code>-r ruta</code>	<i>ruta</i> es un fichero con permiso de lectura (podemos acceder para leer).
<code>-w ruta</code>	<i>ruta</i> es un fichero con permiso de escritura (podemos acceder para escribir).
<code>-x ruta</code>	<i>ruta</i> es un fichero con permiso de ejecución (podemos ejecutarlo).
<code>-f ruta</code>	<i>ruta</i> es un fichero normal (no un directorio, enlace, dispositivo, etc.).
<code>-d ruta</code>	<i>ruta</i> es un directorio.
<code>-b ruta</code>	<i>ruta</i> es un dispositivo de bloques.
<code>-c ruta</code>	<i>ruta</i> es un dispositivo de caracteres.
<code>-L ruta</code>	<i>ruta</i> es un enlace simbólico.
<code>-s ruta</code>	<i>ruta</i> es un fichero con tamaño mayor que cero.
<code>-O ruta</code>	<i>ruta</i> es un fichero cuyo propietario somos nosotros.
<code>-G ruta</code>	<i>ruta</i> es un fichero cuyo grupo es el nuestro.
<code>ruta1 -ot ruta2</code>	<i>ruta1</i> es un fichero más antiguo que <i>ruta2</i> .
<code>ruta1 -nt ruta2</code>	<i>ruta1</i> es un fichero más reciente que <i>ruta2</i> .
<code>-z cadena</code>	<i>cadena</i> es una cadena de caracteres de longitud cero.
<code>-n cadena</code>	<i>cadena</i> es una cadena de caracteres de longitud mayor que cero.
<code>cadena1 = cadena2</code>	<i>cadena1</i> es una cadena igual a <i>cadena2</i> .
<code>cadena1 != cadena2</code>	<i>cadena1</i> es una cadena diferente de <i>cadena2</i> .
<code>número1 -eq número2</code>	<i>número1</i> es un número entero igual a <i>número2</i> .
<code>número1 -ne número2</code>	<i>número1</i> es un número entero diferente de <i>número2</i> .
<code>número1 -gt número2</code>	<i>número1</i> es un número entero mayor que <i>número2</i> .
<code>número1 -ge número2</code>	<i>número1</i> es un número entero mayor o igual que <i>número2</i> .
<code>número1 -lt número2</code>	<i>número1</i> es un número entero menor que <i>número2</i> .
<code>número1 -le número2</code>	<i>número1</i> es un número entero menor o igual que <i>número2</i> .

Operador	Significado
!	Negación.
-a	AND lógico.
-o	OR lógico.
()	Permiten especificar orden de precedencia (hay que utilizarlos con \ o con comillas para eliminar su significado especial para el shell)

15.7.- EL FILTRO *TEE*

El comando tee transmite la entrada estándar a la salida estándar, a la vez que realiza una copia de la entrada en un fichero. Por defecto, se sobrescribe en los ficheros ya existentes.

Si quiere obtener más de una copia, se puede listar más de un fichero de destino.

Opciones: -a Añade la salida a un fichero en lugar de escribirla sobre dicho fichero.

15.8.- LA ORDEN *TRAP*

Esta orden permite especificar una secuencia de ordenes que se ejecutan cuando se reciben señales de interrupción en los programas shell.

Sintaxis

trap { *orden; orden; ...; orden* } señales

- El primer argumento de trap se toma como la orden u ordenes a ejecutar. Si se tienen que ejecutar una secuencia de ordenes se han de incluir entre comillas simples
- Las señales son los códigos de las interrupciones. Las más importantes son:
 - 0 → Salida del shell.
 - 1 → Colgar
 - 2 Interrupción, DELL
 - 3 → Salida.
 - 9 → Matar (No puede ser interrumpida).
 - 15 → Terminación (Kill normal).

```
trap 'cat .ch_history>>$home/ordenes.his; rm .sh_history' 0 1 3 15
{Posibles salidas del sistema}
```


15.9.- EL FILTRO *tr*

El comando *tr* copia los caracteres desde la entrada estándar a la salida estándar sustituyendo o borrando algunos de ellos en el camino. Este comando no puede leer y escribir ficheros, por lo que normalmente se emplearán redirecciones o tuberías para conectar la entrada y salida estándar a otros ficheros o programas. Aquí *cadena1* consta de los caracteres que van a sustituirse y *cadena2* de los caracteres de sustitución.

Sintaxis

tr [-c] [*cadena1* [*cadena2*]]

Operador	Significado
-c	Usa todos los caracteres que no estén en la <i>cadena1</i> , es decir, se toman los caracteres cuyos códigos están comprendidos entre los números octales 001 y 377 y se hace la lista de sustituciones de aquellos que no se den explícitamente en la <i>cadena1</i> .
-d	Los caracteres de <i>cadena1</i> del texto de entrada antes de enviarlos a la salida
-s	Cuando un carácter de <i>cadena1</i> aparece dos o más veces consecutivas en la segunda, comprime la secuencia de caracteres repetidos, hasta dejarla con un solo carácter.

Scripts. Programación del Shell

Los shell de UNIX, además de permitir el uso de comandos, permiten crear programas formados a base de comandos, llamados de muchas formas: shell-scripts, scripts, guiones, etc. Se parecen a los ficheros de proceso por lotes (batch) de MS-DOS (ficheros.bat), pero son mucho más potentes. Además de los comandos, en los scripts podemos utilizar muchas de las estructuras típicas de la programación tradicional: estructuras iterativas, estructuras de control, etc. Los scripts se comportan como auténticos programas (comandos), y admiten parámetros para obtener resultados más completos. Algunos de estos scripts tienen utilidades específicas.*

16.1.- CREACIÓN DE SCRIPTS

Un *script* es, esencialmente, un archivo de texto que contiene en cada línea un comando, una línea en blanco, o un comentario. Los comentarios comienzan por el carácter #, y se pueden incluir en cualquier parte de una línea de un script. Para poder ejecutar un *script* es necesario que establezcamos previamente el permiso de ejecución del fichero correspondiente.

IMPORTANTE:

El shell puede determinar si un fichero es o no de texto examinando la primera línea del fichero; si ésta contiene únicamente caracteres de texto, considerará que todo el fichero es de texto y tratará de ejecutarlo como un script. Sin embargo, si aparece algún carácter que no sea puramente de texto, intentará ejecutarlo como un programa compilado (binario). Esto nos puede ocurrir si en la primera línea de un script escribimos caracteres *extraños*, como vocales acentuadas, eñes, símbolos especiales, etc,

El siguiente ejemplo es un script que podemos utilizar para conocer el estado del sistema: nos muestra la fecha, un calendario y el listado de usuarios conectados. Guardaremos el fichero con el nombre `ejemplo_basico`:

```
date      # MUESTRA LA FECHA
cal       # MUESTRA UN CALENDARIO DE ESTE MES
who       # MUESTRA LA LISTA DE USUARIOS CONECTADOS
```

ejemplo_basico

El resultado de su ejecución es el siguiente:

```
Remigio:~$ ./ejemplo_basico
Tue Oct 26 10:05:15 CEST 1999
    October 1999
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
hans      ttyrc    Oct 26 09:52 (192.168.1.14)
javier    ttyr4    Oct 26 09:49 (192.168.1.29)
amg       ttyr9    Oct 26 09:51 (192.168.1.19)
jupamodo  ttyrd    Oct 26 10:00 (192.168.1.11)
```

Habitualmente en los scripts se utilizan variables para hacerlos más flexibles. Además del uso de las variables que ya conocemos, tenemos nuevas posibilidades al utilizar scripts:

- Lectura del valor de una variable desde teclado.
- Utilización de parámetros.
- Variables especiales.

Para leer un valor por teclado utilizamos el comando **read**, con la siguiente sintaxis:

```
read variable
```

SINTAXIS

variable es el nombre de una variable (sin \$). Por ejemplo:

```
echo Escriba algo:
read ALGO
echo Ha escrito lo siguiente: \"$ALGO\"
```

ejemplo_read

```
Remigio:~$ ./ejemplo_read
Escriba algo:
En un lugar de la mancha...
Ha escrito lo siguiente: "En un lugar de la mancha..."
```

La utilización de parámetros la estudiaremos en el punto siguiente. Por otro lado, las siguientes variables están presentes cuando utilizamos scripts:

- **\$!**: contiene el valor del PID del último proceso ejecutado en segundo plano. Por ejemplo:

```
sleep 3 &      # Ejecuta algo en segundo plano
echo El proceso anterior tiene PID $!
ps             # Muestra los PID de todos los procesos, para poder comparar
```

ejemplo_pid

```
Remigio:~$ ./ejemplo_pid
El proceso anterior tiene PID 7515
  PID TTY STAT  TIME COMMAND
6650  r8 S    0:00 -bash
7514  r8 S    0:00 -bash
7515  r8 S    0:00 sleep 3
7516  r8 R    0:00 ps
```

- **\$?**: contiene el valor de salida del último comando ejecutado. Cuando un programa termina, termina con un estado, que es un número. Si el programa termina de forma normal, este valor será cero. Y si termina con algún tipo de error, el valor devuelto será distinto de cero. Muchos comandos utilizan este mecanismo. La única forma de conocer el valor de retorno de un comando es mediante el uso de esta variable. Ejemplo:

```
test 0 -eq 1    # Debe terminar con error
echo $?
test 0 -eq 0    # Debe terminar correctamente
echo $?
```

ejemplo_test

```
Remigio:~$ ./ejemplo_test
1
0
```

En realidad estas variables también se pueden utilizar desde la línea de comandos (sin script), cuando escribimos varios comandos en la misma orden.

Podemos hacer que nuestros scripts terminen con un estado de normalidad o de error. El comando `exit` (que hasta ahora nos ha servido para salir del sistema) permite terminar prematuramente un script, con un estado determinado. La sintaxis es:

```
exit [estado]
```

SINTAXIS

estado es un número que indica el estado de finalización, que debe ser 0 para representar un estado correcto y un valor distinto de cero para un estado de error (normalmente cada comando utiliza el número de estado erróneo para indicar un tipo concreto de error). Si no se indica estado, se presupone 0. El siguiente ejemplo termina con error de tipo 17:

ejemplo_exit

```

echo Hola
exit 17
echo Esto nunca se escribe!!!

```

Si lo ejecutamos y a continuación analizamos el código de error, observamos que la parte final del programa no se ejecuta, y que terminó con estado 17:

```

Remigio:~$ ./ejemplo_exit ; echo $?
Hola
17

```

16.2.- PARÁMETROS

Los parámetros pasados como argumentos a un script pueden ser consultados como si fueran variables con los nombres de la tabla siguiente.

Variable	Significado
\$1	Primer parámetro pasado.
\$2	Segundo parámetro pasado.
\$3	Tercer parámetro pasado.
...	
\${10}	Décimo parámetro pasado.
\${11}	Undécimo parámetro pasado.
...	
\$*	Todos los parámetros pasados (separados por el primer carácter de la variable IFS).
\$0	Nombre del archivo ejecutado (ruta completa).
\$#	Número de parámetros pasados.

Si consultamos el valor de un parámetro no pasado (por ejemplo, \$4 cuando sólo hay un parámetro), el valor devuelto será una cadena vacía.

Para utilizar los parámetros con dos o más cifras numéricas (es decir, \$10, \$11, \$12, etc.) tenemos que encerrar entre llaves el número (es decir, \${10}, \${11}, \${12}, etc.), ya que de lo contrario se confundirán con alguno de los 10 primeros parámetros seguidos de algún número (es decir, lo equivalente a \${1}0, \${1}1, \${1}2, etc.)

El comando **shift** nos permite desechar el parámetro primero (\$1) y desplazar una posición el resto de parámetros, de manera que \$2 pasa a ocupar la posición de \$1, \$3 la de \$2, y así sucesivamente. Esto nos permite utilizar los parámetros con las estructuras de control iterativas.

Veamos un ejemplo de uso de los parámetros y el comando `shift`. Sea el siguiente script, que llamaremos `ejemplo_par`:

```
echo Hay $# argumentos, que son: $*
echo Argumentos 1, 2 y 3:
echo $1 $2 $3
shift
echo Argumentos 2, 3 y 4:
echo $1 $2 $3
```

ejemplo_par

La siguiente salida muestra un ejemplo de su ejecución:

```
Remigio:~$ ./ejemplo_par A B C D E
Hay 5 argumentos, que son: A B C D E
Argumentos 1, 2 y 3:
A B C
Argumentos 2, 3 y 4:
B C D
```

El número máximo de argumentos es 10. Desde \$0 hasta \$9. A partir de estos argumentos podremos hacer la gestión estadística de archivos.

16.2.1.- Tipos de variables

- Variables asociadas al control del terminal

\$TERM Tipología del terminal, para edición.
\$COLUMNS N° de columnas.
\$LINES N° de líneas.

- Variables asociadas a la gestión de directorio

\$HOME Directorio de trabajo origen.
\$PATH Trayectoria de búsqueda de ejecución.
\$CPATH Trayectoria de directorios sobre los que se accede (cd libros, busca libros dentro de todos los directorios indicados en CPATH)

- Variables asociadas al login

\$LOGNAME Login con el que se accede al sistema

- Variables asociadas a la gestión de correo electrónico

\$MAILPATH Trayectorias de nuestro sistema de correo.
\$MAIL Trayectorias de nuestro sistema de correo.
\$MAILCHECK N° de segundos de test de correo para verificar si hemos recibido.

- Variables asociadas al archivo histórico

\$HISTFILE Indica la trayectoria donde se ubica el archivo histórico.
\$HISTSIZE N° máximo de órdenes permitidas en \$HISTFILE.

- Variables asociadas al editor

\$EDITOR Editor por defecto.

- Variables asociadas al entorno shell

\$SHELL Tipo de shell.

- Variables asociadas al prompt del sistema

\$PS1 Símbolo de petición de orden.

\$PS2 Símbolo de petición de orden secundario.

- Variables asociadas a la medición del tiempo

\$TZ Time zone, referencia la zona horaria de nuestro sistema.

- Otras

\$IFS Separador de campos {`:`, `\n`, `\t`}, controla las delimitaciones de estructuras de archivos en UNIX.

\$ERRNO código numérico de error generado por el proceso.

\$PPID identificador del proceso padre del que está en ejecución.

\$PWD directorio de trabajo

\$SECONDS segundos que llevamos conectados a la sesión.

\$RANDOM generador de valores enteros corto (16 bits) aleatorios

16.3.- SENTENCIAS DE CONTROL

Dentro de un script podemos utilizar algunas de las estructuras habituales de los lenguajes de programación imperativos: estructuras iterativas (bucles *for*, *while*, etc.) y estructuras del control (alternativa, alternativa múltiple, etc.).

16.3.1.- Estructura iterativa *for*

Permite ejecutar una secuencia de comandos varias veces, y en cada una de ellas asigna un valor a una variable, tomado de un conjunto de posibles valores.

```
for variable in conjunto
do
    comandos
done
```

SINTAXIS

variable es el nombre de una variable (sin \$). *conjunto* es un conjunto de valores que se irán asignando a la variable en cada iteración. Podemos utilizar para definir este conjunto máscaras de nombres de ficheros (p.e. *.txt), conjuntos de

valores (p.e. \$*), o una mezcla de ambas cosas (p.e. *.txt \$1 fichero1 palabra1 /tmp/*.c). Finalmente *comandos* es la secuencia de líneas con comandos que se ejecutarán en cada iteración. Un ejemplo: mostrar sucesivamente el contenido los archivos pasados como parámetro, haciendo una pausa entre fichero y fichero:

```
for I in $*
do
    clear
    echo Contenido del fichero $I
    echo -----
    cat $I
    sleep 1
done
```

ejemplo_for

Entre los comandos encerrados por el bucle (entre *do* y *done*) se puede incluir alguna de las siguientes sentencias:

- **break**: permite romper el bucle, de manera que la ejecución continúa por la instrucción inmediatamente posterior al *done* más interno (igual que el *break* de C).
- **continue**: permite iniciar la siguiente iteración de forma inmediata (sin necesidad de llegar al *done*), de forma que el próximo comando que se ejecuta es la asignación de un nuevo valor a la variable de control (igual que el *continue* de C).

16.3.2.- Estructura iterativa *while*

Permite repetir la ejecución de una secuencia de comandos mientras se dé una condición.

```
while condición
do
    comandos
done
```

SINTAXIS

condición es un comando que decide si se continúa iterando. Si el comando termina con éxito, se continúa iterando; por el contrario, si termina con error, termina el bucle. *comandos* es la secuencia de líneas con comandos que se ejecutarán en cada iteración. Veamos un ejemplo: escribir los números del 1 al 30:

```
NUM=1
while [ $NUM -le 30 ]
do
```

ejemplo_while


```
echo $NUM
NUM=`expr $NUM + 1`
done
```

Existen dos comandos muy útiles para ser utilizados en este tipo de bucles. Se trata de los comandos **true** y **false**:

- **true** es un comando que no hace nada, pero que termina con un estado de éxito (0).
- **false** tampoco hace nada, pero termina con un estado de error (distinto de 0).

Por lo tanto, podemos utilizar estos comandos como expresión que siempre se evalúa a verdadero (**true**) o a falso (**false**). Esto permite, entre otras muchas posibilidades, crear bucles infinitos.

En la estructura **while** también podemos utilizar las sentencias **break** y **continue**:

- **break**: permite romper el bucle, de manera que la ejecución continúa por la instrucción inmediatamente posterior al **done** más interno (igual que el **break** de C).
- **continue**: permite iniciar la siguiente iteración de forma inmediata (sin necesidad de llegar al **done**), de forma que el próximo comando que se ejecuta es la evaluación de la condición de continuación (igual que el **continue** de C).

16.3.3.- Estructura iterativa until

Permite repetir la ejecución de una secuencia de comandos hasta que se dé una condición.

```
until condición
do
    comandos
done
```

SINTAXIS

El funcionamiento es idéntico al de la estructura **while**, excepto por que la condición de salida del bucle es contraria a la de éste. Podemos verlo en el ejemplo de uso de **if** (a continuación).

16.3.4.- Estructura alternativa simple (if)

Permite bifurcar la ejecución de dos ramas alternativas, según el valor de una condición.

```

if condición
then
    comandos
[else
    comandos_alternativos]
fi

```

condición es un comando que decide qué rama se ejecuta, la *then*, o la *else*. Si el comando termina con éxito, se ejecutan los comandos de la rama *then*, por el contrario, si termina con error, se ejecutan los comandos de la rama *else*.

comandos es la secuencia de líneas con comandos que se ejecutarán en la rama *then*. *comandos_alternativos* es la secuencia de la rama *else*. La parte *else* es opcional.

Veamos un ejemplo: el juego de adivinar el número de palabras de un fichero pasado por parámetro:

```

TAMANO=`wc -w <$1` # Numero de palabras
NUMERO=-1          # Iniciamos de manera que entre en el bucle

until [ $TAMANO -eq $NUMERO ]
do
    echo Adivina el numero de palabras del fichero $1
    read NUMERO
    if [ $NUMERO -gt $TAMANO ]
    then
        echo El fichero $1 tiene menos palabras.
    else
        if [ $NUMERO -lt $TAMANO ]
        then
            echo El fichero $1 tiene mas palabras.
        fi
    fi
done
echo HAS ACERTADO!!!!

```

16.3.5.- Estructura alternativa múltiple (case)

Permite bifurcar la ejecución entre varias ramas alternativas, según el valor de una expresión.

```

case expresion in
    valor1) comandos1
            comandos1;;
    [ valor2) comandos2
            comandos2;;]
    ...
    [ *)      otros_comandos
            otros_comandos;;]
esac

```

expresión es un comando que decide qué rama se ejecuta. Este comando debe escribir por la salida estándar un valor que será comparado con cada una de las opciones *valor1*, *valor2*, etc. En el caso de que coincida con alguno de estas opciones, ejecutará los comandos situados a la derecha de ")" y en las líneas sucesivas hasta encontrar ";;". Si no coincide con ninguno de los valores propuestos, y existe una opción *), entonces ejecuta los comandos asociados a dicha opción.

Cada valor propuesto puede ser un único valor o un conjunto de valores, separados por "|", y entonces se entrará en la opción cuando la expresión coincida con cualquiera de los valores.

El siguiente ejemplo muestra un menú de opciones para manejar un fichero pasado como parámetro. Se puede mostrar información, su contenido, borrarlo, etc.

ejemplo_case

```
while true
do
    echo OPCIONES
    echo -----
    echo L - listar información del fichero $1
    echo V - visualizar contenido del fichero $1
    echo E - editar fichero $1
    echo R - borrar fichero $1
    echo A - Abortar
    echo
    echo -n "Selecciona una opción: "
    read OPCION
    clear
    case $OPCION in
        L) ls -l $1;;
        E) vi $1;;
        V) more $1;;
        R) rm -f $1
           break;;
        A) break;;
        l|e|r|a|v) echo Debes escribir la opción en mayúsculas.;;
        *) echo Esa opción no existe;;
    esac
done
```

16.3.6.- La orden *xargs*

Una de las características más utilizadas de la programación es la de conectar la salida de un programa con la entrada de otro utilizando cauces. A veces se utiliza la salida de una orden para definir los argumentos de otra

Es una herramienta de programación shell que permite combinar la entrada de una tubería o cauce con la entrada de parámetros posicionales de un shell script.

Sintaxis

xargs [*indicadores*] [orden (*argumentos iniciales*)]

Toma los argumentos iniciales , los combina con los argumentos leídos desde la entrada estándar y utiliza la combinación para ejecutar la orden especificada.

Ejemplos,

```
$echo $* | xargs -n2 -p diff { Argumentos de dos en dos}
```

```
$ls $1 | xargs -i -p mv $1() $2()
```

16.4.- DEPURACIÓN DE *SCRIPTS*

Cuando se está ejecutando un script, no podemos ver la secuencia de comandos que se ejecuta, a menos que activemos la opción `-v` del `bash`, utilizando para ello el comando `set` . Esto nos puede resultar muy útil para saber qué es lo que realmente está ocurriendo en el script.

Veamos por ejemplo la ejecución del siguiente script utilizando la opción `-v`:

```
I=$1
while [ $I -gt 0 ]
do
    echo $I
    I=`expr $I - 1`
done
```

script cuenta

```
Remigio:~$ set -v
Remigio:~$ cuenta 5
cuenta 5
I=$1
while [ $I -gt 0 ]
do
    echo $I
    I=`expr $I - 1`
done
5
expr $I - 1
4
expr $I - 1
3
expr $I - 1
2
expr $I - 1
1
expr $I - 1
```

En lugar de `-v`, podemos utilizar `-x`, que nos muestra lo que realmente se ejecuta en cada instrucción:

```
Remigio:~$ set +v
set +v
Remigio:~$ set -x
Remigio:~$ cuenta 5
+ cuenta 5
++ I=5
++ [ 5 -gt 0 ]
++ echo 5
5
+++ expr 5 - 1
++ I=4
++ [ 4 -gt 0 ]
++ echo 4
4
+++ expr 4 - 1
++ I=3
++ [ 3 -gt 0 ]
++ echo 3
3
+++ expr 3 - 1
++ I=2
++ [ 2 -gt 0 ]
++ echo 2
2
+++ expr 2 - 1
++ I=1
++ [ 1 -gt 0 ]
++ echo 1
1
+++ expr 1 - 1
++ I=0
++ [ 0 -gt 0 ]
```

16.5.- *SCRIPTS* DE INICIACIÓN

Existen algunos scripts que se ejecutan de forma automática cada vez que entramos en el sistema. Se suelen utilizar para definir algunas variables importantes (como el *path*) y ejecutar comandos de iniciación. La Tabla 1 recoge algunos de ellos. Los que están situados en los directorios *home* de los usuarios son personalizables por cada usuario; sin embargo, lo que tiene rutas absolutas son compartidos por todos los usuarios.

Fichero	Utilidad
<code>~/.bash_profile</code>	Se ejecuta cada vez que entra en el sistema, teniendo bash como shell predefinido.
<code>~/.bash_login</code>	Se ejecuta cada vez que entra en el sistema, teniendo bash como shell predefinido, siempre y cuando no exista <code>~/.bash_profile</code> .
<code>~/.profile</code>	Se ejecuta cada vez que entra en el sistema, con cualquier shell predefinido (si el shell es bash, sólo se ejecuta <code>.profile</code> si no existen <code>~/.bash_profile</code> ni <code>~/.bash_login</code>).
<code>/etc/profile</code>	Se ejecuta cada vez que entra en el sistema, teniendo bash como shell predefinido.
<code>~/.bashrc</code>	Se ejecuta cada vez que se ejecuta bash como un comando (sin parámetros) o cuando se entra como otro usuario con <code>su</code> .

Tabla 1. Scripts de iniciación

Bloque III

Seguridad en UNIX

Introducción a la Seguridad en UNIX

En este apartado nos dedicaremos a ver y analizar los diferentes sistemas y las diferentes posibilidades que tenemos para proteger nuestro sistema de posibles agresiones externas o internas. De cualquier forma, será una aproximación bastante superficial, porque no es el asunto central de este manual, cuyo objetivo es recordarnos, una aproximación al sistema operativo UNIX.

16.1.- INTRODUCCIÓN

Hasta finales de 1988 muy poca gente tomaba en serio el tema de la seguridad en redes de computadores de propósito general. Mientras que por una parte Internet iba creciendo exponencialmente con redes importantes que se adherían a ella, como BITNET o HEPNET, por otra el auge de la informática de consumo (hasta la década de los ochenta muy poca gente se podía permitir un ordenador y un módem en casa) unido a factores menos técnicos (como la película *Juegos de Guerra*, de 1983) iba produciendo un aumento espectacular en el número de piratas informáticos. Sin embargo, el 22 de noviembre de 1988 Robert T. Morris protagonizó el primer gran incidente de la seguridad informática: uno de sus programas se convirtió en el famoso *worm* o gusano de Internet. Miles de ordenadores conectados a la red se vieron inutilizados durante días, y las pérdidas se estiman en millones de dólares.

Desde ese momento el tema de la seguridad en sistemas operativos y redes ha sido un factor a tener muy en cuenta por cualquier responsable o administrador de sistemas informáticos. Poco después de este incidente, y a la vista de los potenciales peligros que podía entrañar un fallo o un ataque a los sistemas informáticos estadounidenses (en general, a los sistemas de cualquier país) la agencia DARPA (*Defense Advanced Research Projects Agency*) creó el CERT (*Computer Emergency Response Team*), un grupo formado en su mayor parte por voluntarios cualificados de la comunidad informática, cuyo objetivo principal es facilitar una respuesta rápida a los problemas de seguridad que afecten a *hosts* de Internet ([Den90]).

Han pasado más de diez años desde la creación del primer CERT, y cada día se hace patente la preocupación por los temas relativos a la seguridad en la red y sus equipos, y también se hace patente la necesidad de esta seguridad. Los piratas

de antaño casi han desaparecido, dando paso a nuevas generaciones de intrusos que forman grupos como *Chaos Computer Club* o *Legion of Doom*, organizan encuentros como el español *Iberhack*, y editan revistas o zines electrónicos (*2600: The Hacker's Quartely* o *Phrack* son quizás las más conocidas, pero no las únicas). Todo esto con un objetivo principal: compartir conocimientos. Si hace unos años cualquiera que quisiera adentrarse en el mundo *underground* casi no tenía más remedio que conectar a alguna BBS donde se tratara el tema, generalmente con una cantidad de información muy limitada, hoy en día tiene a su disposición gigabytes de información electrónica publicada en Internet; cualquier aprendiz de pirata puede conectarse a un servidor *web*, descargar un par de programas y ejecutarlos contra un servidor desprotegido... con un poco de (mala) suerte, esa misma persona puede conseguir un control total sobre un servidor UNIX de varios millones de pesetas, probablemente desde su PC con Windows 98 y sin saber nada sobre UNIX. De la misma forma que en su día *Juegos de Guerra* creó una nueva generación de piratas, en la segunda mitad de los noventa películas como *The Net*, *Hackers* o *Los Corsarios del Chip* han creado otra generación, en general mucho menos peligrosa que la anterior, pero cuanto menos, preocupante: aunque sin grandes conocimientos técnicos, tienen a su disposición multitud de programas y documentos sobre seguridad (algo que los piratas de los ochenta apenas podían imaginar), además de ordenadores potentes y conexiones a Internet baratas. Por si esto fuera poco, se ven envalentonados a través de sistemas de conversación como el IRC (*Internet Relay Chat*), donde en canales como *#hack* o *#hackers* presumen de sus logros ante sus colegas.

16.2.- ¿QUÉ ES LA SEGURIDAD?

Podemos entender como **seguridad** una característica de cualquier sistema (informático o no) que nos indica que ese sistema está libre de todo peligro, daño o riesgo, y que es, en cierta manera, infalible. Como esta característica, particularizando para el caso de sistemas operativos o redes de computadores, es muy difícil de conseguir (según la mayoría de expertos, imposible), se suaviza la definición de **seguridad** y se pasa a hablar de **fiabilidad** (probabilidad de que un sistema se comporte tal y como se espera de él) más que de **seguridad**, por tanto, se habla de **sistemas fiables** en lugar de hacerlo de **sistemas seguros**.

A grandes rasgos se entiende que mantener un sistema **seguro** (o fiable) consiste básicamente en garantizar tres aspectos: **confidencialidad**, **integridad** y **disponibilidad**.

¿Qué implica cada uno de los tres aspectos de los que hablamos? La **confidencialidad** nos dice que los objetos de un sistema han de ser accedidos únicamente por elementos autorizados a ello, y que esos elementos autorizados no van a convertir esa información en disponible para otras entidades; la **integridad** significa que los objetos sólo pueden ser modificados por elementos autorizados, y de una manera controlada, y la **disponibilidad** indica que los objetos del sistema tienen que permanecer accesibles a elementos autorizados; es el contrario de la negación de servicio. Generalmente tienen que existir los tres aspectos descritos

para que haya seguridad: un sistema UNIX puede conseguir confidencialidad para un determinado fichero haciendo que ningún usuario (ni siquiera el *root*) pueda leerlo, pero este mecanismo no proporciona disponibilidad alguna.

Dependiendo del entorno en que un sistema UNIX trabaje, a sus responsables les interesará dar prioridad a un cierto aspecto de la seguridad. Por ejemplo, en un sistema militar se antepondría la confidencialidad de los datos almacenados o transmitidos sobre su disponibilidad: seguramente, es preferible que alguien borre información confidencial (que se podría recuperar después desde una cinta de *backup*) a que ese mismo atacante pueda leerla, o a que esa información esté disponible en un instante dado para los usuarios autorizados. En cambio, en un servidor NFS de un departamento se premiará la disponibilidad frente a la confidencialidad: importa poco que un atacante lea una unidad, pero que esa misma unidad no sea leída por usuarios autorizados va a suponer una pérdida de tiempo y dinero. En un entorno bancario, la faceta que más ha de preocupar a los responsables del sistema es la integridad de los datos, frente a su disponibilidad o su confidencialidad: es menos grave que un usuario consiga leer el saldo de otro que el hecho de que ese usuario pueda modificarlo.

16.3.- ¿DE QUÉ NOS QUEREMOS PROTEGER?

A continuación se presenta una relación de los elementos que potencialmente pueden amenazar a nuestro sistema. No pretende ser exhaustiva; simplemente trata de proporcionar una idea acerca de qué o quién amenaza un sistema UNIX.

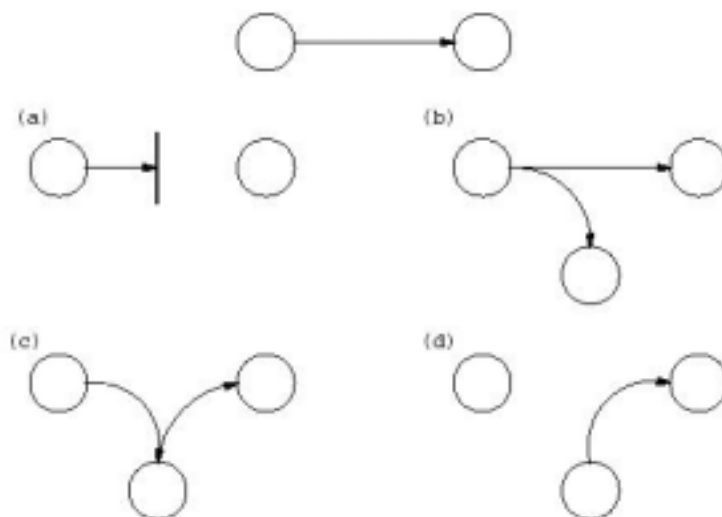


Figura 1.1: Flujo normal de información entre emisor y receptor y posibles amenazas: (a) interrupción, (b) interceptación, (c) modificación y (d) fabricación.

16.3.1.- Personas

No podemos engañarnos: la mayoría de ataques a nuestro sistema van a provenir en última instancia de personas que, intencionada o inintencionadamente, pueden causarnos enormes pérdidas. Generalmente se tratará de piratas que intentan conseguir el máximo nivel de privilegio posible aprovechando alguno (o algunos) de los riesgos lógicos de los que hablaremos a continuación, especialmente agujeros del *software*. Pero con demasiada frecuencia se suele olvidar que los piratas ‘clásicos’ no son los únicos que amenazan nuestros equipos: es especialmente preocupante que mientras que hoy en día cualquier administrador mínimamente preocupado por la seguridad va a conseguir un sistema relativamente fiable de una forma lógica (permaneciendo atento a vulnerabilidades de su *software*, restringiendo servicios, utilizando cifrado de datos. . .), pocos administradores tienen en cuenta factores como la ingeniería social o el basureo a la hora de diseñar una política de seguridad.

* **Curiosos**

Junto con los *crackers*, los curiosos son los atacantes más habituales de sistemas UNIX en redes de I+D. Recordemos que los equipos están trabajando en entornos donde se forma a futuros profesionales de la informática y las telecomunicaciones (gente que *a priori* tiene interés por las nuevas tecnologías), y recordemos también que las personas suelen ser curiosas por naturaleza; esta combinación produce una avalancha de estudiantes o personal intentando conseguir mayor privilegio del que tienen o intentando acceder a sistemas a los que oficialmente no tienen acceso.

Y en la mayoría de ocasiones esto se hace simplemente para leer el correo de un amigo, enterarse de cuánto cobra un compañero, copiar un trabajo o comprobar que es posible romper la seguridad de un sistema concreto. Aunque en la mayoría de situaciones se trata de ataques no destructivos (a excepción del borrado de huellas para evitar la detección), parece claro que no benefician en absoluto al entorno de fiabilidad que podamos generar en un determinado sistema.

* **Crackers**

Los entornos de seguridad media son un objetivo típico de los intrusos, ya sea para fisgonear, para utilizarlas como enlace hacia otras redes o simplemente por diversión. Por un lado, son redes generalmente abiertas, y la seguridad no es un factor tenido muy en cuenta en ellas; por otro, el gran número y variedad de sistemas UNIX conectados a estas redes provoca, casi por simple probabilidad, que al menos algunos de sus equipos (cuando no la mayoría) sean vulnerables a problemas conocidos de antemano. De esta forma un atacante sólo ha de utilizar un escáner de seguridad contra el dominio completo y luego atacar mediante un simple *exploit* los equipos que presentan vulnerabilidades; esto convierte a las redes de **I+D**, a las de empresas, o a las de **ISPs** en un objetivo fácil y apetecible para piratas con cualquier nivel de conocimientos, desde los más novatos (y a veces más peligrosos) hasta los expertos, que pueden utilizar toda la red para probar nuevos ataques o como nodo intermedio en un ataque a otros organismos, con el consiguiente

deterioro de imagen (y a veces de presupuesto) que supone para una universidad ser, sin desearlo, un apoyo a los piratas que atacan sistemas teóricamente más protegidos, como los militares.

***Terroristas**

Por ‘terroristas’ no debemos entender simplemente a los que se dedican a poner bombas o quemar autobuses; bajo esta definición se engloba a cualquier persona que ataca al sistema simplemente por causar algún tipo de daño en él. Por ejemplo, alguien puede intentar borrar las bases de datos de un partido político enemigo o destruir los sistemas de ficheros de un servidor que alberga páginas *web* de algún grupo religioso; en el caso de redes de **I+D**, típicos ataques son la destrucción de sistemas de prácticas o la modificación de páginas *web* de algún departamento o de ciertos profesores, generalmente por parte de alumnos descontentos.

***Intrusos remunerados**

Este es el grupo de atacantes de un sistema más peligroso, aunque por fortuna el menos habitual en redes normales; suele afectar más a las grandes – muy grandes – empresas o a organismos de defensa. Se trata de piratas con gran experiencia en problemas de seguridad y un amplio conocimiento del sistema, que son pagados por una tercera parte⁵ generalmente para robar secretos (el nuevo diseño de un procesador, una base de datos de clientes, información confidencial sobre las posiciones de satélites espía. . .) o simplemente para dañar la imagen de la entidad afectada. Esta tercera parte suele ser una empresa de la competencia o un organismo de inteligencia, es decir, una organización que puede permitirse un gran gasto en el ataque; de ahí su peligrosidad: se suele pagar bien a los mejores piratas, y por si esto fuera poco los atacantes van a tener todos los medios necesarios a su alcance.

Aunque como hemos dicho los intrusos remunerados son los menos comunes en la mayoría de situaciones, en ciertas circunstancias pueden aprovechar nuestras redes como plataforma para atacar otros organismos; una excelente lectura sobre esta situación es en la que el experto en seguridad Cli Stoll describe cómo piratas pagados por el KGB soviético utilizaron redes y sistemas UNIX dedicados a **I+D** para acceder a organismos de defensa e inteligencia estadounidenses.

16.3.2.- Amenazas Lógicas

Bajo la etiqueta de ‘amenazas lógicas’ encontramos todo tipo de programas que de una forma u otra pueden dañar a nuestro sistema, creados de forma intencionada para ello (*software* malicioso, también conocido como *malware*) o simplemente por error (*bugs* o agujeros).

*** Software incorrecto**

Las amenazas más habituales a un sistema UNIX provienen de errores cometidos de forma involuntaria por los programadores de sistemas o de aplicaciones. Una situación no contemplada a la hora de diseñar el sistema de red del *kernel* o un error accediendo a memoria en un fichero *setuidado* pueden

comprometer local o remotamente a UNIX (o a cualquier otro sistema operativo).

A estos errores de programación se les denomina *bugs*, y a los programas utilizados para aprovechar uno de estos fallos y atacar al sistema, *exploits*. Como hemos dicho, representan la amenaza más común contra UNIX, ya que cualquiera puede conseguir un *exploit* y utilizarlo contra nuestra máquina sin ni siquiera saber cómo funciona y sin unos conocimientos mínimos de UNIX; incluso hay *exploits* que dañan seriamente la integridad de un sistema (**negaciones de servicio**[DOS] o incluso acceso **root** remoto) y están preparados para ser utilizados desde MS-DOS, con lo que cualquier pirata novato (comúnmente, se les denomina *Script Kiddies*) puede utilizarlos contra un servidor y conseguir un control total de una máquina de varios millones de pesetas desde su PC sin saber nada del sistema atacado; incluso hay situaciones en las que se analizan los *logs* de estos ataques y se descubre que el pirata incluso intenta ejecutar órdenes de MS-DOS.

*** Herramientas de seguridad**

Cualquier herramienta de seguridad representa un arma de doble filo: de la misma forma que un administrador las utiliza para detectar y solucionar fallos en sus sistemas o en la subred completa, un potencial intruso las puede utilizar para detectar esos mismos fallos y aprovecharlos para atacar los equipos. Herramientas como Nessus, Saint o Satán pasan de ser útiles a ser peligrosas cuando las utilizan *crackers* que buscan información sobre las vulnerabilidades de un *host* o de una red completa.

La conveniencia de diseñar y distribuir libremente herramientas que puedan facilitar un ataque es un tema peliagudo; incluso expertos reconocidos como Alec Muet (autor del adivinador de contraseñas **Crack**) han recibido enormes críticas por diseñar determinadas herramientas de seguridad para UNIX. Tras numerosos debates sobre el tema, ha quedado bastante claro que no se puede basar la seguridad de un sistema en el supuesto desconocimiento de sus problemas por parte de los atacantes: esta política, denominada *Security through obscurity*, se ha demostrado inservible en múltiples ocasiones. Si como administradores no utilizamos herramientas de seguridad que muestren las debilidades de nuestros sistemas (para corregirlas), tenemos que estar seguro que un atacante no va a dudar en utilizar tales herramientas (para explotar las debilidades encontradas); por tanto, hemos de agradecer a los diseñadores de tales programas el esfuerzo que han realizado (y nos han ahorrado) en pro de sistemas más seguros.

*** Puertas traseras**

Durante el desarrollo de aplicaciones grandes o de sistemas operativos es habitual entre los programadores insertar ‘atajos’ en los sistemas habituales de autenticación del programa o del núcleo que se está diseñando. A estos atajos se les denomina **puertas traseras**, y con ellos se consigue mayor velocidad a la hora de detectar y depurar fallos: por ejemplo, los diseñadores de un *software* de gestión de bases de datos en el que para acceder a una tabla se necesitan cuatro claves diferentes de diez caracteres cada una pueden insertar una rutina para conseguir ese acceso mediante una única clave ‘especial’, con el objetivo de perder menos tiempo al depurar el sistema.

Algunos programadores pueden dejar estos atajos en las versiones definitivas de su *software* para facilitar un mantenimiento posterior, para garantizar su propio acceso, o simplemente por descuido; la cuestión es que si un atacante descubre una de estas puertas traseras (no nos importa el método que utilice para hacerlo) va a tener un acceso global a datos que no debería poder leer, lo que obviamente supone un grave peligro para la integridad de nuestro sistema.

***Bombas lógicas**

Las bombas lógicas son partes de código de ciertos programas que permanecen sin realizar ninguna función hasta que son activadas; en ese punto, la función que realizan no es la original del programa, sino que generalmente se trata de una acción perjudicial.

Los activadores más comunes de estas bombas lógicas pueden ser la ausencia o presencia de ciertos ficheros, la ejecución bajo un determinado UID o la llegada de una fecha concreta; cuando la bomba se activa va a poder realizar cualquier tarea que pueda realizar la persona que ejecuta el programa: si las activa el *root*, o el programa que contiene la bomba está *setuidado* a su nombre, los efectos obviamente pueden ser fatales.

***Canales cubiertos**

Son canales de comunicación que permiten a un proceso transferir información de forma que viole la política de seguridad del sistema; dicho de otra forma, un proceso transmite información a otros (locales o remotos) que no están autorizados a leer dicha información.

Los canales cubiertos no son una amenaza demasiado habitual en redes de **I+D**, ya que suele ser mucho más fácil para un atacante aprovechar cualquier otro mecanismo de ataque lógico; sin embargo, es posible su existencia, y en este caso su detección suele ser difícil: algo tan simple como el puerto *finger* abierto en una máquina puede ser utilizado a modo de *covert channel* por un pirata con algo de experiencia.

*** Virus**

Un virus es una secuencia de código que se inserta en un fichero ejecutable (denominado *huésped*), de forma que cuando el archivo se ejecuta, el virus también lo hace, insertándose a sí mismo en otros programas.

Todo el mundo conoce los efectos de los virus en algunos sistemas operativos de sobremesa; sin embargo, en UNIX los virus no suelen ser un problema de seguridad grave, ya que lo que pueda hacer un virus lo puede hacer más fácilmente cualquier otro mecanismo lógico (que será el que hay que tener en cuenta a la hora de diseñar una política de seguridad).

Aunque los virus existentes para entornos UNIX son más una curiosidad que una amenaza real, en sistemas sobre plataformas IBM-PC o compatibles (recordemos que hay muchos sistemas UNIX que operan en estas plataformas, como UNIX, FreeBSD, NetBSD, Minix, Solaris. . .) ciertos virus, especialmente los de *boot*, pueden tener efectos nocivos, como dañar el sector de arranque; aunque se trata de daños menores comparados con los efectos de otras amenazas, hay que tenerlos en cuenta.

* **Gusanos**

Un gusano es un programa capaz de ejecutarse y propagarse por sí mismo a través de redes, en ocasiones portando virus o aprovechando *bugs* de los sistemas a los que conecta para dañarlos.

Al ser difíciles de programar su número no es muy elevado, pero el daño que pueden causar es muy grande: el mayor incidente de seguridad en Internet fue precisamente el *Internet Worm*, un gusano que en 1988 causó pérdidas millonarias al infectar y detener más de 6000 máquinas conectadas a la red.

Hemos de pensar que un gusano puede automatizar y ejecutar en unos segundos todos los pasos que seguiría un atacante humano para acceder a nuestro sistema: mientras que una persona, por muchos conocimientos y medios que posea, tardaría como mínimo horas en controlar nuestra red completa (un tiempo más que razonable para detectarlo), un gusano puede hacer eso mismo en pocos minutos: de ahí su enorme peligro y sus devastadores efectos.

* **Caballos de Troya**

Los troyanos o caballos de Troya son instrucciones escondidas en un programa de forma que éste parezca realizar las tareas que un usuario espera de él, pero que realmente ejecute funciones ocultas (generalmente en detrimento de la seguridad) sin el conocimiento del usuario; como el Caballo de Troya de la mitología griega, al que deben su nombre, ocultan su función real bajo la apariencia de un programa inofensivo que a primera vista funciona correctamente.

En la práctica totalidad de los ataques a UNIX, cuando un intruso consigue el privilegio necesario en el sistema instala troyanos para ocultar su presencia o para asegurarse la entrada en caso de ser descubierto: por ejemplo, es típico utilizar lo que se denomina un *rootkit*, que no es más que un conjunto de versiones troyanas de ciertas utilidades (**netstat**, **ps**, **who**. . .), para conseguir que cuando el administrador las ejecute no vea la información relativa al atacante, como sus procesos o su conexión al sistema; otro programa que se suele suplantar es *login*, por ejemplo para que al recibir un cierto nombre de usuario y contraseña proporcione acceso al sistema sin necesidad de consultar **/etc/passwd**.

* **Programas conejo o bacterias**

Bajo este nombre se conoce a los programas que no hacen nada útil, sino que simplemente se dedican a reproducirse hasta que el número de copias acaba con los recursos del sistema (memoria, procesador, disco. . .), produciendo una negación de servicio. Por sí mismos no hacen ningún daño, sino que lo que realmente perjudica es el gran número de copias suyas en el sistema, que en algunas situaciones pueden llegar a provocar la parada total de la máquina.

Hemos de pensar hay ciertos programas que pueden actuar como conejos sin proponérselo; ejemplos típicos se suelen encontrar en los sistemas UNIX destinados a prácticas en las que se enseña a programar al alumnado: es muy común que un bucle que por error se convierte en infinito contenga entre sus instrucciones algunas de reserva de memoria, lo que implica que si el

sistema no presenta una correcta política de cuotas para procesos de usuario pueda venirse abajo o degradar enormemente sus prestaciones. El hecho de que el autor suela ser fácilmente localizable no debe ser ninguna excusa para descuidar esta política: no podemos culpar a un usuario por un simple error, y además el daño ya se ha producido.

* Técnicas salami

Por técnica salami se conoce al robo automatizado de pequeñas cantidades de bienes (generalmente dinero) de una gran cantidad origen. El hecho de que la cantidad inicial sea grande y la robada pequeña hace extremadamente difícil su detección: si de una cuenta con varios millones de pesetas se roban unos céntimos, nadie va a darse cuenta de ello; si esto se automatiza para, por ejemplo, descontar una peseta de cada nómina pagada en la universidad o de cada beca concedida, tras un mes de actividad seguramente se habrá robado una enorme cantidad de dinero sin que nadie se haya percatado de este hecho, ya que de cada origen se ha tomado una cantidad ínfima.

Las técnicas salami no se suelen utilizar para atacar sistemas normales, sino que su uso más habitual es en sistemas bancarios; sin embargo, como en una red con requerimientos de seguridad medios es posible que haya ordenadores dedicados a contabilidad, facturación de un departamento o gestión de nóminas del personal, comentamos esta potencial amenaza contra el *software* encargado de estas tareas.

16.4.- ¿CÓMO NOS PODEMOS PROTEGER?

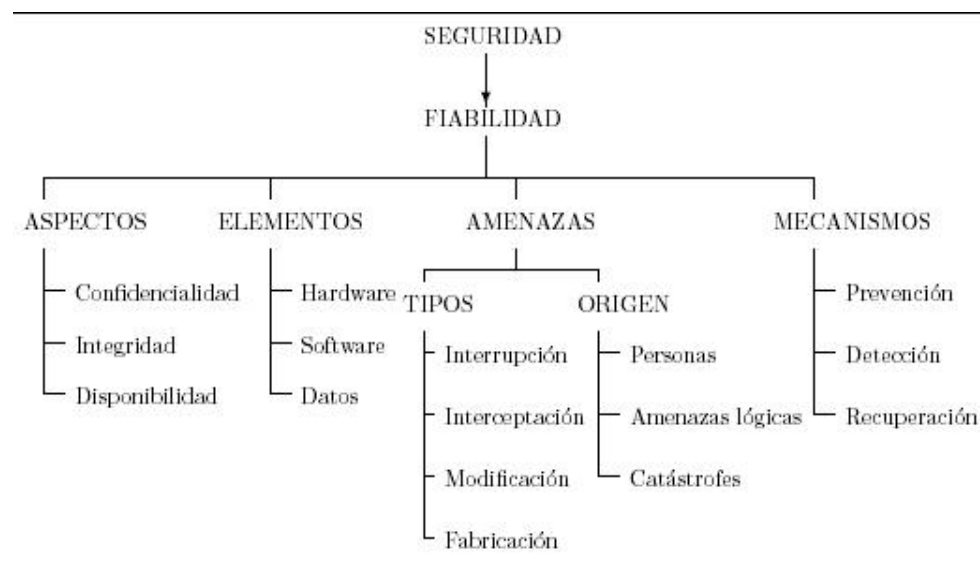


Figura 1.2: Visión global de la seguridad informática

Los mecanismos de prevención más habituales en UNIX y en redes son los siguientes:

Mecanismos de autenticación e identificación

Estos mecanismos hacen posible identificar entidades del sistema de una forma única, y posteriormente, una vez identificadas, autenticarlas (comprobar que la entidad es quien dice ser). Son los mecanismos más importantes en cualquier sistema, ya que forman la base de otros mecanismos que basan su funcionamiento en la identidad de las entidades que acceden a un objeto.

Un grupo especialmente importante de estos mecanismos son los denominados Sistemas de Autenticación de Usuarios, a los que prestaremos una especial atención por ser los más utilizados en la práctica.

Mecanismos de control de acceso

Cualquier objeto del sistema ha de estar protegido mediante mecanismos de control de acceso, que controlan todos los tipos de acceso sobre el objeto por parte de cualquier entidad del sistema. Dentro de UNIX, el control de acceso más habitual es el **discrecional** (DAC, *Discretionary Access Control*), implementado por los bits *rwX* y las listas de control de acceso para cada fichero (objeto) del sistema; sin embargo, también se permiten especificar controles de acceso obligatorio (MAC).

Mecanismos de separación

Cualquier sistema con diferentes niveles de seguridad ha de implementar mecanismos que permitan separar los objetos dentro de cada nivel, evitando el flujo de información entre objetos y entidades de diferentes niveles siempre que no exista una autorización expresa del mecanismo de control de acceso.

Los mecanismos de separación se dividen en cinco grandes grupos, en función de como separan a los objetos: separación física, temporal, lógica, criptográfica y fragmentación. Dentro de UNIX, el mecanismo de separación más habitual es el de separación lógica o **aislamiento**, implementado en algunos sistemas mediante una **Base Segura de Cómputo** (TCB).

Mecanismos de seguridad en las comunicaciones

Es especialmente importante para la seguridad de nuestro sistema el proteger la integridad y la privacidad de los datos cuando se transmiten a través de la red. Para garantizar esta seguridad en las comunicaciones, hemos de utilizar ciertos mecanismos, la mayoría de los cuales se basan en la Criptografía: cifrado de clave pública, de clave privada, firmas digitales. . . Aunque cada vez se utilizan más los protocolos seguros (como **SSH** o *Kerberos*, en el caso de sistemas UNIX en red), aún es frecuente encontrar conexiones en texto claro ya no sólo entre máquinas de una misma subred, sino entre redes diferentes. Una de las mayores amenazas a la integridad de las redes es este tráfico sin cifrar, que hace extremadamente fáciles ataques encaminados a robar contraseñas o suplantar la identidad de máquinas de la red.

La seguridad de los ficheros

Anteriormente, en la sección dedicada a las órdenes y manejo de UNIX, hemos visto cómo funcionan los permisos en los ficheros y directorios y las restricciones que dichos permisos crean para evitar un acceso no-autorizado. Aquí profundizaremos en el aspecto de la seguridad de los ficheros.

18.1.- BITS *SUID*, *SGID* Y *STICKY*

Habitualmente, los permisos de los archivos en Unix se corresponden con un número en octal que varía entre 000 y 777; sin embargo, existen unos permisos especiales que hacen variar ese número entre 0000 y 7777: se trata de los bits de permanencia (1000), **sgid** (2000) y **suid** (4000).

El bit de **suid** o **setuid** se activa sobre un fichero añadiéndole 4000 a la representación octal de los permisos del archivo y otorgándole además permiso de ejecución al propietario del mismo; al hacer esto, en lugar de la **x** en la primera terna de los permisos, aparecerá una **s** o una **S** si no hemos otorgado el permiso de ejecución correspondiente (en este caso el bit no tiene efecto):

```
anita:~# chmod 4777 /tmp/file1

anita:~# chmod 4444 /tmp/file2

anita:~# ls -l /tmp/file1

-rwsrwxrwx 1 root other 0 Feb 9 17:51 /tmp/file1*

anita:~# ls -l /tmp/file2

-r-Sr--r-- 1 root other 0 Feb 9 17:51 /tmp/file2*

anita:~#
```

El bit **suid** activado sobre un fichero indica que todo aquél que ejecute el archivo va a tener durante la ejecución los mismos privilegios que quién lo creó; dicho de otra forma, si el administrador crea un fichero y lo **setuیدا**, todo aquel

usuario que lo ejecute va a disponer, hasta que el programa finalice, de un nivel de privilegio total en el sistema. Podemos verlo con el siguiente ejemplo:

```
luisa:/home/toni# cat testsuid.c

#include <stdio.h>

#include <unistd.h>

main(){

printf("UID: %d, EUID: %d\n",getuid(),geteuid());

}

luisa:/home/toni# cc -o testsuid testsuid.c

luisa:/home/toni# chmod u+s testsuid

luisa:/home/toni# ls -l testsuid

-rwsr-xr-x 1 root root 4305 Feb 10 02:34 testsuid

luisa:/home/toni# su toni

luisa:~$ id

uid=1000(toni) gid=100(users) groups=100(users)

luisa:~$ ./testsuid

UID: 1000, EUID: 0

luisa:~$
```

Podemos comprobar que el usuario *toni*, sin ningún privilegio especial en el sistema, cuando ejecuta nuestro programa *setuidado* de prueba está trabajando con un **euid** (*Effective UID*) 0, lo que le otorga todo el poder del administrador (fijémonos que éste último es el propietario del ejecutable); si en lugar de este código el ejecutable fuera una copia de un *shell*, el usuario *toni* tendría todos los privilegios del *root* mientras no finalice la ejecución, es decir, hasta que no se teclee **exit** en la línea de órdenes.

Todo lo que acabamos de comentar con respecto al bit *setuid* es aplicable al bit *setgid* pero a nivel de grupo del fichero en lugar de propietario: en lugar de trabajar con el **euid** del propietario, todo usuario que ejecute un programa *setgidado* tendrá los privilegios del grupo al que pertenece el archivo. Para activar el bit de *setgid* sumaremos 2000 a la representación octal del permiso del fichero y además habremos de darle permiso de ejecución a la terna de grupo; si lo hacemos, la **s** o **S** aparecerá en lugar de la **x** en esta terna. Si el fichero es un directorio y no un archivo plano, el bit *setgid* afecta a los ficheros y subdirectorios que se creen en él: estos tendrán como grupo propietario al mismo que el directorio *setgidado*, siempre que el proceso que los cree pertenezca a dicho grupo.

Pero, ¿cómo afecta todo esto a la seguridad del sistema? Muy sencillo: los bits de **setuid** y **setgid** dan a Unix una gran flexibilidad, pero constituyen al mismo tiempo la mayor fuente de ataques internos al sistema (entendiendo por ataques internos aquellos realizados por un usuario – autorizado o no – desde la propia máquina, generalmente con el objetivo de aumentar su nivel de privilegio en la misma). Cualquier sistema Unix tiene un cierto número de ejecutables **setuidados** y/o **setgidados**. Cada uno de ellos, como acabamos de comentar, se ejecuta con los privilegios de quien lo creó (generalmente el **root** u otro usuario con ciertos privilegios) lo que directamente implica que cualquier usuario tiene la capacidad de lanzar tareas que escapen total o parcialmente al control del sistema operativo: se ejecutan en modo privilegiado si es el administrador quien creó los ejecutables. Evidentemente, estas tareas han de estar controladas de una forma exhaustiva, ya que si una de ellas se comporta de forma anormal (un simple **core dump**) puede causar daños irreparables al sistema; tanto es así que hay innumerables documentos que definen, o lo intentan, pautas de programación considerada ‘segura’. Si por cualquier motivo un programa **setuidado** falla se asume inmediatamente que presenta un problema de seguridad para la máquina, y se recomienda resetear el bit de **seguid** cuanto antes.

Está claro que asegurar completamente el comportamiento correcto de un programa es muy difícil, por no decir imposible; cada cierto tiempo suelen aparecer fallos (**bugs**) en ficheros **setuidados** de los diferentes clones de Unix que ponen en peligro la integridad del sistema. Entonces, ¿por qué no se adopta una solución radical, como eliminar este tipo de archivos? Hay una sencilla razón: el riesgo que presentan no se corre inútilmente, para tentar al azar, sino que los archivos que se ejecutan con privilegios son estrictamente necesarios en Unix, al menos algunos de ellos. Veamos un ejemplo: un fichero **setuidado** clásico en cualquier clon es **/bin/passwd**, la orden para que los usuarios puedan cambiar su contraseña de entrada al sistema. No hace falta analizar con mucho detalle el funcionamiento de este programa para darse cuenta que una de sus funciones consiste en modificar el fichero de claves (**/etc/passwd** o **/etc/shadow**). Está claro que un usuario **per se** no tiene el nivel de privilegio necesario para hacer esto (incluso es posible que ni siquiera pueda leer el fichero de claves), por lo que frente a este problema tan simple existen varias soluciones: podemos asignar permiso de escritura para todo el mundo al fichero de contraseñas, podemos denegar a los usuarios el cambio de clave o podemos obligarles a pasar por la sala de operaciones cada vez que quieran cambiar su contraseña. Parece obvio que ninguna de ellas es apropiada para la seguridad del sistema (quizás la última lo sea, pero es impracticable en máquinas con un número de usuarios considerable). Por tanto, debemos asumir que el bit de **setuid** en **/bin/passwd** es imprescindible para un correcto funcionamiento del sistema. Sin embargo, esto no siempre sucede así en un sistema Unix instalado **out of the box** el número de ficheros **setuidados** suele ser mayor de cincuenta; sin perjudicar al correcto funcionamiento de la máquina, este número se puede reducir a menos de cinco, lo que viene a indicar que una de las tareas de un administrador sobre un sistema recién instalado es minimizar el número de ficheros **setuidados** o **setgidados**. No obstante, tampoco es conveniente eliminarlos, sino simplemente resetear su bit de **setuid** mediante **chmod**:

```

luisa:~# ls -l /bin/ping

-r-sr-xr-x 1 root bin 14064 May 10 1999 /bin/ping*

luisa:~# chmod -s /bin/ping

luisa:~# ls -l /bin/ping

-r-xr-xr-x 1 root bin 14064 May 10 1999 /bin/ping*

luisa:~#

```

También hemos de estar atentos a nuevos ficheros de estas características que se localicen en la máquina; demasiadas aplicaciones de Unix se instalan por defecto con ejecutables *setuidados* cuando realmente este bit no es necesario, por lo que a la hora de instalar nuevo *software* o actualizar el existente hemos de acordarnos de resetear el bit de los ficheros que no lo necesiten. Especialmente grave es la aparición de archivos *setuidados* de los que el administrador no tenía constancia (ni son aplicaciones del sistema ni un aplicaciones añadidas), ya que esto casi en el 100% de los casos indica que nuestra máquina ha sido comprometida por un atacante. Para localizar los ficheros con alguno de estos bits activos, podemos ejecutar la siguiente orden:

```

luisa:~# find / \( -perm -4000 -o -perm -2000 \) -type f -print

```

Por otra parte, el *sticky bit* o bit de permanencia se activa sumándole 1000 a la representación octal de los permisos de un determinado archivo y otorgándole además permiso de ejecución; si hacemos esto, veremos que en lugar de una **X** en la terna correspondiente al resto de usuarios aparece una **t** (si no le hemos dado permiso de ejecución al archivo, aparecerá una **T**):

```

anita:~# chmod 1777 /tmp/file1

anita:~# chmod 1774 /tmp/file2

anita:~# ls -l /tmp/file1

-rwxrwxrwt 1 root other 0 Feb 9 17:51 /tmp/file1*

anita:~# ls -l /tmp/file2

-rwxrwxr-T 1 root other 0 Feb 9 17:51 /tmp/file2*

anita:~#

```

Si el bit de permanencia de un fichero está activado (recordemos que si aparece una **T** no lo está) le estamos indicando al sistema operativo que se trata de un archivo muy utilizado, por lo que es conveniente que permanezca en memoria principal el mayor tiempo posible; esta opción se utilizaba en sistemas antiguos que disponían de muy poca RAM, pero hoy en día prácticamente no se utiliza. Lo que si que sigue vigente es el efecto del *sticky bit* activado sobre un directorio: en este caso se indica al sistema operativo que, aunque los permisos ‘normales’ digan que cualquier usuario pueda crear y eliminar ficheros (por ejemplo, un 777 octal), sólo

el propietario de cierto archivo y el administrador pueden borrar un archivo guardado en un directorio con estas características. Este bit, que sólo tiene efecto cuando es activado por el administrador (aunque cualquier usuario puede hacer que aparezca una **t** o una **T** en sus ficheros y directorios), se utiliza principalmente en directorios del sistema de ficheros en los que interesa que todos puedan escribir pero que no todos puedan borrar los datos escritos, como `/tmp/` o `/var/tmp/`: si el equivalente octal de los permisos de estos directorios fuera simplemente 777 en lugar de 1777, cualquier usuario podría borrar los ficheros del resto. Si pensamos que para evitar problemas podemos simplemente denegar la escritura en directorios como los anteriores también estamos equivocados: muchos programas – como compiladores, editores o gestores de correo – asumen que van a poder crear ficheros en `/tmp/` o `/var/tmp/`, de forma que si no se permite a los usuarios hacerlo no van a funcionar correctamente; por tanto, es muy recomendable para el buen funcionamiento del sistema que al menos el directorio `/tmp/` tenga el bit de permanencia activado.

Ya para finalizar, volvemos a lo que hemos comentado al principio de la sección: el equivalente octal de los permisos en Unix puede variar entre 0000 y 7777. Hemos visto que podíamos sumar 4000, 2000 o 1000 a los permisos ‘normales’ para activar respectivamente los bits *setuid*, *setgid* o *sticky*. Por supuesto, podemos activar varios de ellos a la vez simplemente sumando sus valores: en la situación poco probable de que necesitáramos todos los bits activos, sumaríamos 7000 a la terna octal 777:

```
luisa:~# chmod 0 /tmp/fichero

luisa:~# ls -l /tmp/fichero

----- 1 root root 0 Feb 9 05:05 /tmp/fichero

luisa:~# chmod 7777 /tmp/fichero

luisa:~# ls -l /tmp/fichero

-rwsrwsrwt 1 root root 0 Feb 9 05:05 /tmp/fichero*

luisa:~#
```

Si en lugar de especificar el valor octal de los permisos queremos utilizar la forma simbólica de **chmod**, utilizaremos **+t** para activar el bit de permanencia, **g+s** para activar el de *setgid* y **u+s** para hacer lo mismo con el de *setuid*; si queremos resetearlos, utilizamos un signo `-` en lugar de un `+` en la línea de órdenes.

18.2.- ALMACENAMIENTO SEGURO

En esta sección abordaremos las diferentes formas para poder guardar nuestros datos y enviarlos de forma que nuestra confidencialidad no se vea amenazada.

18.2.1.- La orden *crypt*

La orden **crypt** permite cifrar y descifrar ficheros en diferentes sistemas Unix; si no recibe parámetros lee los datos de la entrada estándar y los escribe en la salida estándar, por lo que seguramente habremos de redirigir ambas a los nombres de fichero adecuados. Un ejemplo simple de su uso puede ser el siguiente:

```
$ crypt <fichero.txt >fichero.crypt
```

4.8. ALMACENAMIENTO SEGURO 63

Enter key:

```
$
```

En el anterior ejemplo hemos cifrado utilizando **crypt** el archivo **fichero.txt** y guardado el resultado en **fichero.crypt**; el original en texto claro se mantiene en nuestro directorio, por lo que si queremos evitar que alguien lo lea deberemos borrarlo. Para descifrar un fichero cifrado mediante **crypt** (por ejemplo, el anterior) utilizamos la misma orden y la misma clave:

```
$ crypt <fichero.crypt>salida.txt
```

Enter key:

```
$
```

El anterior comando ha descifrado **fichero.crypt** con la clave tecleada y guardado el resultado en el archivo **salida.txt**, que coincidirá en contenido con el anterior **fichero.txt**. **crypt** no se debe utilizar **nunca** para cifrar información confidencial; la seguridad del algoritmo de cifra utilizado por esta orden es mínima, ya que **crypt** se basa en una máquina con un rotor de 256 elementos similar en muchos aspectos a la alemana *Enigma*, con unos métodos de ataque rápidos y conocidos por todos. Por si esto fuera poco, si en lugar de teclear la clave cuando la orden nos lo solicita lo hacemos en línea de comandos, como en el siguiente ejemplo:

```
$ crypt clave < fichero.txt > fichero.crypt
```

```
$
```

Entonces a la debilidad criptográfica de **crypt** se une el hecho de que en muchos Unixes cualquier usuario puede observar la clave con una orden tan simple como **ps** (no obstante, para minimizar este riesgo, el propio programa guarda la clave y la elimina de su línea de argumentos nada más leerla).

Obviamente, la orden **crypt** no tiene nada que ver con la función **crypt**, utilizada a la hora de cifrar claves de usuarios, que está basada en una variante del algoritmo **des** y se puede considerar segura en la mayoría de entornos.

18.2.2.- PGP: *Pretty Good Privacy*

El **software** PGP, desarrollado por el criptólogo estadounidense Phil Zimmermann, es mundialmente conocido como sistema de firma digital para correo electrónico. Aparte de esta función, PGP permite también el cifrado de archivos de forma convencional mediante criptografía simétrica; esta faceta de PGP convierte a este programa en una excelente herramienta para cifrar archivos que almacenamos en nuestro sistema; no es el mismo mecanismo que el que se emplea para cifrar un fichero que vamos a enviar por correo, algo que hay que hacer utilizando la clave pública del destinatario, sino que es un método que no utiliza para nada los anillos de PGP, los **userID** o el cifrado asimétrico. Para ello utilizamos la opción **-c** desde línea de órdenes:

```
anita:~$ pgp -c fichero.txt

No configuration file found.

Pretty Good Privacy(tm) 2.6.3i - Public-key encryption for the masses.

(c) 1990-96 Philip Zimmermann, Phil's Pretty Good Software. 1996-01-18

International version - not for use in the USA. Does not use RSAREF.

Current time: 2000/03/02 07:18 GMT

You need a pass phrase to encrypt the file.

Enter pass phrase:

Enter same pass phrase again:

Preparing random session key...Just a moment....

Ciphertext file: fichero.txt.pgp

anita:~$
```

Esta orden nos preguntará una clave para cifrar, una **pass phrase**, que no tiene por qué ser (ni es recomendable que lo sea) la misma que utilizamos para proteger la clave privada, utilizada en el sistema de firma digital. A partir de la clave tecleada (que obviamente no se muestra en pantalla), PGP generará un archivo denominado **fichero.txt.pgp** cuyo contenido es el resultado de comprimir y cifrar (en este orden) el archivo original. Obviamente, **fichero.txt** no se elimina automáticamente, por lo que es probable que deseemos borrarlo a mano.

Si lo que queremos es obtener el texto en claro de un archivo previamente cifrado simplemente hemos de pasar como parámetro el nombre de dicho fichero:

```
anita:~$ pgp fichero.txt.pgp

No configuration file found.

Pretty Good Privacy(tm) 2.6.3i - Public-key encryption for the masses.

(c) 1990-96 Philip Zimmermann, Phil's Pretty Good Software. 1996-01-18
```


International version - not for use in the USA. Does not use RSAREF.

Current time: 2000/03/02 07:24 GMT

File is conventionally encrypted.

You need a pass phrase to decrypt this file.

Enter pass phrase:

Just a moment....Pass phrase appears good. .

Plaintext filename: fichero.txt

anita:~\$

Como vemos, se nos pregunta la clave que habíamos utilizado para cifrar el archivo, y si es correcta se crea el fichero con el texto en claro; como sucedía antes, el archivo original no se elimina, por lo que tendremos ambos en nuestro directorio.

PGP ofrece un nivel de seguridad muy superior al de **crypt**, ya que utiliza algoritmos de cifra más robustos: en lugar de implementar un modelo similar a Enigma, basado en máquinas de rotor, PGP ofrece cifrado simétrico principalmente mediante IDEA, un algoritmo de clave secreta desarrollado a finales de los ochenta por Xuejia Lai y James Massey. Aparte de IDEA, en versiones posteriores a la utilizada aquí se ofrecen también Triple DES (similar a DES pero con una longitud de clave mayor) y CAST5, un algoritmo canadiense que hasta la fecha sólo ha podido ser atacado con éxito mediante fuerza bruta; este último es el cifrador simétrico utilizado por defecto en PGP 5.x.

18.2.3.- TFCS: *Transparent Cryptographic File System*

TCFS es un **software** desarrollado en la Universidad de Salerno y disponible para sistemas Linux que proporciona una solución al problema de la privacidad en sistemas de ficheros distribuidos como NFS: típicamente en estos entornos las comunicaciones se realizan en texto claro, con la enorme amenaza a la seguridad que esto implica. TCFS almacena los ficheros cifrados, y son pasados a texto claro antes de ser leídos; todo el proceso se realiza en la máquina cliente, por lo que las claves nunca son enviadas a través de la red.

La principal diferencia de TCFS con respecto a otros sistemas de ficheros cifrados como CFS es que, mientras que éstos operan a nivel de aplicación, TCFS lo hace a nivel de núcleo, consiguiendo así una mayor transparencia y seguridad. Obviamente esto tiene un grave inconveniente: TCFS sólo está diseñado para funcionar dentro del núcleo de sistemas Linux, por lo que si nuestra red de Unix utiliza otro clon del sistema operativo, no podremos utilizar TCFS correctamente. No obstante, esta gran integración de los servicios de cifrado en el sistema de los ficheros hace que el modelo sea transparente al usuario final.

Para utilizar TCFS necesitamos que la máquina que exporta directorios vía NFS ejecute el demonio **xattrd**; por su parte, los clientes han de ejecutar un núcleo compilado con soporte para TCFS.

Además, el administrador de la máquina cliente ha de autorizar a los usuarios a que utilicen TCFS, generando una clave que cada uno de ellos utilizará para trabajar con los ficheros cifrados; esto se consigue mediante **tcfskeygen**, que genera una entrada para cada usuario en **/etc/tcfspasswd**:

```
rosita:~# tcfskeygen

login: toni

password:

now we'll generate the des key.

press 10 keys:*****

Ok.

rosita:~# cat /etc/tcfspasswd

toni:2rCmyOUsm5IA=

rosita:~#
```

Una vez que un usuario tiene una entrada en **/etc/tcfspasswd** con su clave ya puede acceder a ficheros cifrados; para ello, en primer lugar utilizará **tcfslogin** para insertar su clave en el **kernel**, tras lo cual puede ejecutar la variante de **mount** distribuida con TCFS para montar los sistemas que el servidor exporta. Sobre los archivos de estos sistemas, se utiliza la variante de **chattr** de TCFS para activar o desactivar el atributo **X** (podemos visualizar los atributos de un fichero con **lsattr**), que indica que se trata de archivos que necesitan al demonio de TCFS para trabajar sobre ellos (cifrando o descifrando). Finalmente, antes de abandonar una sesión se ha de ejecutar **tcfslogout**, cuya función es eliminar la clave del **kernel** de Linux. También es necesaria una variante de **passwd**, proporcionada con TCFS, que regenera las claves de acceso a archivos cifrados cuando un usuario cambia su **password**.

TCFS utiliza uno de los cuatro modos de funcionamiento que ofrece el estándar DES denominado CBC (*Cipher Block Chaining*). El principal problema de este modelo (aparte de la potencial inseguridad de DES) es la facilidad para insertar información al final del fichero cifrado, por lo que es indispensable recurrir a estructuras que permitan detectar el final real de cada archivo; otro problema, menos peligroso *a priori*, es la repetición de patrones en archivos que ocupen más de 34 Gigabytes (aproximadamente), que puede conducir, aunque es poco probable, a un criptoanálisis exitoso en base a estas repeticiones. Más peligroso es el uso del mismo **password** de entrada al sistema como clave de cifrado utilizando la función resumen MD5 (el peligro no proviene del uso de esta

función *hash*, sino de la clave del usuario); previsiblemente en futuras versiones de CFS se utilizarán *passphrases* similares a las de PGP para descifrar y descifrar.

Auditoría del sistema

En UNIX tan importante es mantener un control acerca de las actividades del sistema, como poder acceder a una serie de recursos para comprobar qué servicios de nuestro sistema están siendo o han sido modificados. De eso trata este capítulo

19.1.- INTRODUCCIÓN

Casi todas las actividades realizadas en un sistema Unix son susceptibles de ser, en mayor o menor medida, monitorizadas: desde las horas de acceso de cada usuario al sistema hasta las páginas **web** más frecuentemente visitadas, pasando por los intentos fallidos de conexión, los programas ejecutados o incluso el tiempo de CPU que cada usuario consume. Obviamente esta facilidad de Unix para recoger información tiene unas ventajas inmediatas para la seguridad: es posible detectar un intento de ataque nada más producirse el mismo, así como también detectar usos indebidos de los recursos o actividades ‘sospechosas’; sin embargo, existen también desventajas, ya que la gran cantidad de información que potencialmente se registra puede ser aprovechada para crear negaciones de servicio o, más habitualmente, esa cantidad de información puede hacer difícil detectar problemas por el volumen de datos a analizar.

Algo muy interesante de los archivos de **log** en Unix es que la mayoría de ellos son simples ficheros de texto, que se pueden visualizar con un simple **cat**. Por una parte esto es bastante cómodo para el administrador del sistema, ya que no necesita de herramientas especiales para poder revisar los **logs** (aunque existen algunas utilidades para hacerlo, como **swatch**) e incluso puede programar **shellscripts** para comprobar los informes generados de forma automática, con órdenes como **awk**, **grep** o **sed**. No obstante, el hecho de que estos ficheros sean texto plano hace que un atacante lo tenga muy fácil para ocultar ciertos registros modificando los archivos con cualquier editor de textos; esto implica una cosa muy importante para un administrador: **nunca** ha de confiar al 100% en lo que los informes de auditoría del sistema le digan. Para minimizar estos riesgos se pueden tomar diversas medidas, desde algunas quizás demasiado complejas para entornos habituales hasta otras más sencillas pero igualmente efectivas, como utilizar una máquina fiable para registrar información del sistema o incluso enviar los registros más importantes a una impresora; más adelante hablaremos de ellas.

19.2.- EL SISTEMA DE LOG EN UNIX

Una desventaja añadida al sistema de auditoría en Unix puede ser la complejidad que puede alcanzar una correcta configuración; por si la dificultad del sistema no fuera suficiente, en cada Unix el sistema de *logs* tiene peculiaridades que pueden propiciar la pérdida de información interesante de cara al mantenimiento de sistemas seguros. Aunque muchos de los ficheros de *log* de los que hablaremos a continuación son comunes en cualquier sistema, su localización, o incluso su formato, pueden variar entre diferentes Unices.

Dentro de Unix hay dos grandes familias de sistemas: se trata de *System V* y *bsd*; la localización de ficheros y ciertas órdenes relativas a la auditoría de la máquina van a ser diferentes en ellas, por lo que es muy recomendable consultar las páginas del manual antes de ponerse a configurar el sistema de auditoría en un equipo concreto. La principal diferencia entre ellos es el denominado *process accounting* o simplemente *accounting*, consistente en registrar todos los programas ejecutados por cada usuario; evidentemente, los informes generados en este proceso pueden llegar a ocupar muchísimo espacio en disco (dependiendo del número de usuarios en nuestro sistema) por lo que sólo es recomendable en situaciones muy concretas, por ejemplo para detectar actividades sospechosas en una máquina o para cobrar por el tiempo de CPU consumido. En los sistemas *System V* el *process accounting* está desactivado por defecto; se puede iniciar mediante `/usr/lib/acct/startup`, y para visualizar los informes se utiliza la orden `acctcom`. En la familia *bsd* los equivalentes a estas órdenes son `accton` y `lastcomm`; en este caso el *process accounting* está inicializado por defecto.

Un mundo aparte a la hora de generar (y analizar) informes acerca de las actividades realizadas sobre una máquina Unix son los sistemas con el modelo de auditoría C2; mientras que con el modelo clásico se genera un registro tras la ejecución de cada proceso, en Unix C2 se proporciona una pista de auditoría donde se registran los accesos y los intentos de acceso de una entidad a un objeto, así como cada cambio en el estado del objeto, la entidad o el sistema global. Esto se consigue asignando un identificador denominado *Audit ID* a cada grupo de procesos ejecutados (desde el propio *login*), identificador que se registra junto a la mayoría de llamadas al sistema que un proceso realiza, incluyendo algunas tan comunes como `write()`, `open()`, `close()` o `read()`. A nadie se le puede escapar la cantidad de espacio y de CPU necesarios para mantener los registros a un nivel tan preciso, por lo que en la mayoría de sistemas (especialmente en entornos habituales, como los estudiados aquí) el modelo de auditoría C2 es innecesario; y no sólo esto, sino que en muchas ocasiones también se convierte en una monitorización inútil si no se dispone de mecanismos para interpretar o reducir la gran cantidad de datos registrados: el administrador guarda tanta información que es casi imposible analizarla en busca de actividades sospechosas.

19.3.- EL DEMONIO SYSLOGD

El demonio **syslogd** (*Syslog Daemon*) se lanza automáticamente al arrancar un sistema Unix, y es el encargado de guardar informes sobre el funcionamiento de la máquina. Recibe mensajes de las diferentes partes del sistema (núcleo, programas...) y los envía y/o almacena en diferentes localizaciones, tanto locales como remotas, siguiendo un criterio definido en el fichero de configuración **/etc/syslog.conf**, donde especificamos las reglas a seguir para gestionar el almacenamiento de mensajes del sistema. Las líneas de este archivo que comienzan por **`#'** son comentarios, con lo cual son ignoradas de la misma forma que las líneas en blanco; si ocurriera un error al interpretar una de las líneas del fichero, se ignoraría la línea completa. Un ejemplo de fichero **/etc/syslog.conf** es el siguiente:

```
anita:~# cat /etc/syslog.conf
#ident "@(#)syslog.conf 1.4 96/10/11 SMI" /* SunOS 5.0 */
#
# Copyright (c) 1991-1993, by Sun Microsystems, Inc.
#
# syslog configuration file.
#
# This file is processed by m4 so be careful to quote (`') names
# that match m4 reserved words. Also, within ifdef's, arguments
# containing commas must be quoted.
#
*.err;kern.notice;auth.notice /dev/console
*.err;kern.debug;daemon.notice;mail.crit /var/adm/messages
*.alert;kern.err;daemon.err operator
*.alert root
6.3. EL DEMONIO SYSLOGD 93
*.emerg *
# if a non-loghost machine chooses to have authentication messages
# sent to the loghost machine, un-comment out the following line:
#auth.notice ifdef(`LOGHOST', /var/log/authlog, @loghost)
mail.debug ifdef(`LOGHOST', /var/log/syslog, @loghost)
#
# non-loghost machines will use the following lines to cause "user"
# log messages to be logged locally.
#
ifdef(`LOGHOST', ,
user.err /dev/console
user.err /var/adm/messages
user.alert `root, operator'
user.emerg *
)
anita:~#
```

Podemos ver que cada regla del archivo tiene dos campos: un campo de selección y un campo de acción, separados por espacios o tabuladores. El **campo de selección** está formado a su vez de dos partes: una del servicio que envía el mensaje y otra de su prioridad, separadas por un punto (**`.`**); ambas son indiferentes a mayúsculas y minúsculas. La parte del servicio contiene una de las siguientes palabras clave: **auth**, **auth-priv**, **cron**, **daemon**, **kern**, **lpr**, **mail**, **mark**, **news**, **security** (equivalente a **auth**), **syslog**, **user**, **uucp** y **local0** hasta **local7**. Esta parte especifica el 'subsistema' que ha generado ese mensaje (por ejemplo, todos los programas relacionados con el correo generarán mensajes ligados al servicio **mail**). La prioridad está compuesta de uno de los siguientes términos, en orden

ascendente: **debug**, **info**, **notice**, **warning**, **warn** (equivalente a **warning**), **err**, **error** (equivalente a **err**), **crit**, **alert**, **emerg**, y **panic** (equivalente a **emerg**). La prioridad define la gravedad o importancia del mensaje almacenado. Todos los mensajes de la prioridad especificada y superiores son almacenados de acuerdo con la acción requerida. Además de los términos mencionados hasta ahora, el demonio **syslogd** emplea los siguientes caracteres especiales:

- **'*' (asterisco)**

Empleado como comodín para todas las prioridades y servicios anteriores, dependiendo de dónde son usados (si antes o después del carácter de separación ':'):

```
# Guardar todos los mensajes del servicio mail en /var/adm/mail
#
mail.* /var/adm/mail
```

- **' ' (blanco, espacio, nulo)**

Indica que no hay prioridad definida para el servicio de la línea almacenada.

- **',' (coma)**

Con este carácter es posible especificar múltiples servicios con el mismo patrón de prioridad en una misma línea. Es posible enumerar cuantos servicios se quieran:

```
# Guardar todos los mensajes mail.info y news.info en
# /var/adm/info
mail,news.=info /var/adm/info
```

- **',' (punto y coma)**

Es posible dirigir los mensajes de varios servicios y prioridades a un mismo destino, separándolos por este carácter:

```
# Guardamos los mensajes de prioridad "info" y "notice"
# en el archivo /var/log/messages
*.=info;*.=notice /var/log/messages
```

- **'=' (igual)**

De este modo solo se almacenan los mensajes con la prioridad exacta especificada y no incluyendo las superiores:

```
# Guardar todos los mensajes criticos en /var/adm/critical
#
*.=crit /var/adm/critical
```

- **'!' (exclamación)**

Preceder el campo de prioridad con un signo de exclamación sirve para ignorar todas las prioridades, teniendo la posibilidad de escoger entre la especificada (**!=prioridad**) y la especificada más todas las superiores (**!prioridad**). Cuando se usan conjuntamente los caracteres '=' y '!', el signo de exclamación '!' debe preceder obligatoriamente al signo igual '=', de esta forma: **!=**.

```
# Guardar mensajes del kernel de prioridad info, pero no de
# prioridad err y superiores
# Guardar mensajes de mail excepto los de prioridad info
```

```
kern.info;kern.!err /var/adm/kernel-info
mail.*;mail.!=info /var/adm/mail
```

Por su parte, el **campo de acción** describe el destino de los mensajes, que puede ser :

Un fichero plano

Normalmente los mensajes del sistema son almacenados en ficheros planos. Dichos ficheros han de estar especificados con la ruta de acceso completa (comenzando con `/). Podemos preceder cada entrada con el signo menos, '-', para omitir la sincronización del archivo (vaciado del *bufferr* de memoria a disco). Aunque puede ocurrir que se pierda información si el sistema cae justo después de un intento de escritura en el archivo, utilizando este signo se puede conseguir una mejora importante en la velocidad, especialmente si estamos ejecutando programas que mandan muchos mensajes al demonio **syslogd**.

```
# Guardamos todos los mensajes de prioridad critica en "critical"
#
*.=crit /var/adm/critical
```

Un terminal (o la consola)

También tenemos la posibilidad de enviar los mensajes a terminales; de este modo podemos tener uno de los terminales virtuales que muchos sistemas Unix ofrecen en su consola 'dedicado' a listar los mensajes del sistema, que podrán ser consultados con solo cambiar a ese terminal:

```
# Enviamos todos los mensajes a tty12 (ALT+F12 en Linux) y todos
# los mensajes criticos del nucleo a consola
#
*.* /dev/tty12
kern.crit /dev/console
```

Una tubería con nombre

Algunas versiones de **syslogd** permiten enviar registros a ficheros de tipo **pipe** simplemente anteponiendo el símbolo `|` al nombre del archivo; dicho fichero ha de ser creado antes de iniciar el demonio **syslogd**, mediante órdenes como **mknfifo** o **mknod**. Esto es útil para *debug* y también para procesar los registros utilizando cualquier aplicación de Unix, tal y como veremos al hablar de *logs* remotos cifrados.

Por ejemplo, la siguiente línea de **/etc/syslog.conf** enviaría todos los mensajes de cualquier prioridad a uno de estos ficheros denominado **/var/log/mififo**:

```
# Enviamos todos los mensajes a la tuberia con nombre
# /var/log/mififo
#
*.* | /var/log/mififo
```


Una máquina remota

Se pueden enviar los mensajes del sistema a otra máquina, de manera a que sean almacenados remotamente. Esto es útil si tenemos una máquina segura, en la que podemos confiar, conectada a la red, ya que de esta manera se guardaría allí una copia de los mensajes de nuestro sistema y no podrían ser modificados en caso de que alguien entrase en nuestra máquina. Esto es especialmente útil para detectar usuarios ‘ocultos’ en nuestro sistema (usuarios maliciosos que han conseguido los suficientes privilegios para ocultar sus procesos o su conexión):

```
# Enviamos los mensajes de prioridad warning y superiores al
# fichero "syslog" y todos los mensajes (incluidos los
# anteriores) a la maquina "secure.upv.es"
#
*.warn /usr/adm/syslog
*. * @secure.upv.es
```

Unos usuarios del sistema (si están conectados)

Se especifica la lista de usuarios que deben recibir un tipo de mensajes simplemente escribiendo su *login*, separados por comas:

```
# Enviamos los mensajes con la prioridad "alert" a root y toni
#
*.alert root, toni
```

Todos los usuarios que estén conectados

Los errores con una prioridad de emergencia se suelen enviar a todos los usuarios que estén conectados al sistema, de manera que se den cuenta de que algo va mal:

```
# Mostramos los mensajes urgentes a todos los usuarios
# conectados, mediante wall
*.=emerg
```

19.4.- ALGUNOS ARCHIVOS DE LOG

En función de la configuración del sistema de auditoría de cada equipo Unix los eventos que sucedan en la máquina se registrarán en determinados ficheros; aunque podemos *loggear* en cualquier fichero (incluso a través de la red o en dispositivos, como veremos luego), existen ciertos archivos de registro ‘habituales’ en los que se almacena información. A continuación comentamos los más comunes y la información que almacenan.

19.4.1.- *syslog*

El archivo **syslog** (guardado en `/var/adm/` o `/var/log/`) es quizás el fichero de *log* más importante del sistema; en él se guardan, en texto claro, mensajes relativos a la seguridad de la máquina, como los accesos o los intentos de acceso a ciertos servicios. No obstante, este fichero es escrito por **syslogd**, por lo

que dependiendo de nuestro fichero de configuración encontraremos en el archivo una u otra información. Al estar guardado en formato texto, podemos visualizar su contenido con un simple **cat**:

```
anita:/# cat /var/log/syslog
Mar 5 04:15:23 anita in.telnetd[11632]: connect from localhost
Mar 5 06:16:52 anita rpcbind: connect from 127.0.0.1 to getport(R )
Mar 5 06:16:53 anita last message repeated 3 times
Mar 5 06:35:08 anita rpcbind: connect from 127.0.0.1 to getport(R )
Mar 5 18:26:56 anita rpcbind: connect from 127.0.0.1 to getport(R )
Mar 5 18:28:47 anita last message repeated 1 time
Mar 5 18:32:43 anita rpcbind: connect from 127.0.0.1 to getport(R )
Mar 6 02:30:26 anita rpcbind: connect from 127.0.0.1 to getport(R )
Mar 6 03:31:37 anita rpcbind: connect from 127.0.0.1 to getport(R )
Mar 6 11:07:04 anita in.telnetd[14847]: connect from rosita
Mar 6 11:40:43 anita in.telnetd[14964]: connect from localhost
anita:/#
```

19.4.2.- *messages*

En este archivo de texto se almacenan datos ‘informativos’ de ciertos programas, mensajes de baja o media prioridad destinados más a informar que a avisar de sucesos importantes, como información relativa al arranque de la máquina; no obstante, como sucedía con el fichero **syslog**, en función de **/etc/syslog.conf** podremos guardar todo tipo de datos. Para visualizar su contenido es suficiente una orden como **cat** o similares:

```
anita:/# head -70 /var/adm/messages
Jan 24 18:09:54 anita unix: SunOS Release 5.7 Version Generic
[UNIX(R) System V Release 4.0]
Jan 24 18:09:54 anita unix: Copyright (c) 1983-1998, Sun
Microsystems, Inc.
Jan 24 18:09:54 anita unix: mem = 65152K (0x3fa0000)
Jan 24 18:09:54 anita unix: avail mem = 51167232
Jan 24 18:09:54 anita unix: root nexus = i86pc
Jan 24 18:09:54 anita unix: isa0 at root
Jan 24 18:09:54 anita unix: pci0 at root: space 0 offset 0
Jan 24 18:09:54 anita unix: IDE device at targ 0, lun 0 lastlun 0x0
Jan 24 18:09:54 anita unix: model WDC WD84AA, stat 50, err 0
Jan 24 18:09:54 anita unix: cfg 0x427a, cyl 16383, hd 16, sec/trk 63
Jan 24 18:09:54 anita unix: mult1 0x8010, mult2 0x110, dwcap 0x0,
cap 0x2f00
Jan 24 18:09:54 anita unix: piomode 0x280, dmamode 0x0, advpiomode
0x3
Jan 24 18:09:54 anita unix: minpio 120, minpioflow 120
Jan 24 18:09:54 anita unix: valid 0x7, dwdma 0x7, majver 0x1e
Jan 24 18:09:54 anita unix: ata_set_feature: (0x66,0x0) failed
Jan 24 18:09:54 anita unix: ATAPI device at targ 1, lun 0 lastlun
0x0
Jan 24 18:09:54 anita unix: model CD-ROM 50X, stat 50, err 0
Jan 24 18:09:54 anita unix: cfg 0x85a0, cyl 0, hd 0, sec/trk 0
Jan 24 18:09:54 anita unix: mult1 0x0, mult2 0x0, dwcap 0x0, cap
0xf00
Jan 24 18:09:54 anita unix: piomode 0x400, dmamode 0x200, advpiomode
0x3
Jan 24 18:09:54 anita unix: minpio 227, minpioflow 120
Jan 24 18:09:54 anita unix: valid 0x6, dwdma 0x107, majver 0x0
Jan 24 18:09:54 anita unix: PCI-device: ata@0, ata0
```

6.4. ALGUNOS ARCHIVOS DE LOG 97

```
Jan 24 18:09:54 anita unix: ata0 is /pci@0,0/pci-ide@7,1/ata@0
Jan 24 18:09:54 anita unix: Disk0: <Vendor 'Gen-ATA ' Product 'WDC
WD84AA '>
Jan 24 18:09:54 anita unix: cmdk0 at ata0 target 0 lun 0
Jan 24 18:09:54 anita unix: cmdk0 is /pci@0,0/pci-
ide@7,1/ata@0/cmdk@0,0
Jan 24 18:09:54 anita unix: root on /pci@0,0/pci-
ide@7,1/ide@0/cmdk@0,0:a
fstype ufs
Jan 24 18:09:54 anita unix: ISA-device: asy0
Jan 24 18:09:54 anita unix: asy0 is /isa/asy@1,3f8
Jan 24 18:09:54 anita unix: ISA-device: asy1
Jan 24 18:09:54 anita unix: asy1 is /isa/asy@1,2f8
Jan 24 18:09:54 anita unix: ISA-device: asy2
Jan 24 18:09:54 anita unix: asy2 is
/isa/pnpSUP,1670@pnpSUP,1670,7ec2
Jan 24 18:09:54 anita unix: Number of console virtual screens = 13
Jan 24 18:09:54 anita unix: cpu 0 initialization complete - online
Jan 24 18:09:54 anita unix: dump on /dev/dsk/c0d0s1 size 86 MB
Jan 24 18:09:55 anita unix: pseudo-device: pm0
Jan 24 18:09:55 anita unix: pm0 is /pseudo/pm@0
Jan 24 18:09:56 anita unix: pseudo-device: vol0
Jan 24 18:09:56 anita unix: vol0 is /pseudo/vol@0
Jan 24 18:09:57 anita icmpinfo: started, PID=213.
Jan 24 18:09:57 anita unix: sdl at ata0:
Jan 24 18:09:57 anita unix: target 1 lun 0
Jan 24 18:09:57 anita unix: sdl is /pci@0,0/pci-ide@7,1/ata@0/sd@1,0
Jan 24 18:10:03 anita icmpinfo: ICMP_Dest_Unreachable[Port] <
127.0.0.1
[localhost] > 127.0.0.1 [localhost] sp=1664 dp=3200 seq=0x002e0000
sz=74(+20)
Jan 24 18:10:03 anita unix: ISA-device: fdc0
Jan 24 18:10:03 anita unix: fd0 at fdc0
Jan 24 18:10:03 anita unix: fd0 is /isa/fdc@1,3f0/fd@0,0
Jan 24 18:10:04 anita icmpinfo: ICMP_Dest_Unreachable[Port] <
127.0.0.1
[localhost] > 127.0.0.1 [localhost] sp=2944 dp=161 seq=0x00420000
sz=92(+20)
Jan 24 18:10:05 anita unix: ISA-device: asy0
Jan 24 18:10:05 anita unix: asy0 is /isa/asy@1,3f8
Jan 24 18:10:05 anita unix: ISA-device: asy1
Jan 24 18:10:05 anita unix: asy1 is /isa/asy@1,2f8
Jan 24 18:10:05 anita unix: ISA-device: asy2
Jan 24 18:10:05 anita unix: asy2 is
/isa/pnpSUP,1670@pnpSUP,1670,7ec2
Jan 24 18:10:08 anita icmpinfo: ICMP_Dest_Unreachable[Port] <
127.0.0.1
[localhost] > 127.0.0.1 [localhost] sp=32780 dp=162 seq=0x00370000
sz=83(+20)
Jan 24 18:10:35 anita unix: pseudo-device: xsvc0
Jan 24 18:10:35 anita unix: xsvc0 is /pseudo/xsvc@0
anita:/#
```

19.4.3.- wtmp

Este archivo es un fichero binario (no se puede leer su contenido directamente volcándolo con **cat** o similares) que almacena información relativa a cada conexión y desconexión al sistema. Podemos ver su contenido con órdenes como **last**:

```

anita:/# last -10
toni pts/11 localhost Mon Mar 6 11:07 - 11:07 (00:00)
toni pts/11 rosita Sun Mar 5 04:22 - 04:25 (00:03)
ftp ftp andercheran.aiin Sun Mar 5 02:30 still logged in
ftp ftp andercheran.aiin Sun Mar 5 00:28 - 02:30 (02:01)
ftp ftp anita Thu Mar 2 03:02 - 00:28 (2+21:25)
ftp ftp anita Thu Mar 2 03:01 - 03:02 (00:00)
98 CAPÍTULO 6. AUDITORÍA DEL SISTEMA
ftp ftp localhost Thu Mar 2 02:35 - 03:01 (00:26)
root console Thu Mar 2 00:13 still logged in
reboot system boot Thu Mar 2 00:12
root console Wed Mar 1 06:18 - down (17:54)
anita:/#

```

Los registros guardados en este archivo (y también en **utmp**) tienen el formato de la estructura **utmp**, que contiene información como el nombre de usuario, la línea por la que accede, el lugar desde donde lo hace y la hora de acceso; se puede consultar la página de manual de funciones como **getutent()** para ver la estructura concreta en el clon de Unix en el que trabajemos. Algunas variantes de Unix (como Solaris o IRIX) utilizan un fichero **wtmp** extendido denominado **wtmpx**, con campos adicionales que proporcionan más información sobre cada conexión.

19.4.4.- *utmp*

El archivo **utmp** es un fichero binario con información de cada usuario que está conectado en un momento dado; el programa **/bin/login** genera un registro en este fichero cuando un usuario conecta, mientras que **init** lo elimina cuando desconecta. Aunque habitualmente este archivo está situado en **/var/adm/**, junto a otros ficheros de **log**, es posible encontrar algunos Unices – los más antiguos – que lo sitúan en **/etc/**. Para visualizar el contenido de este archivo podemos utilizar órdenes como **last** (indicando el nombre de fichero mediante la opción **-f**), **w** o **who**:

```

anita:/# who
root console Mar 2 00:13
root pts/2 Mar 3 00:47 (unix)
root pts/3 Mar 2 00:18 (unix)
root pts/5 Mar 2 00:56 (unix)
root pts/6 Mar 2 02:23 (unix:0.0)
root pts/8 Mar 3 00:02 (unix:0.0)
root pts/7 Mar 2 23:43 (unix:0.0)
root pts/9 Mar 3 00:51 (unix)
root pts/10 Mar 6 00:23 (unix)
anita:/#

```

Como sucedía con **wtmp**, algunos Unices utilizan también una versión extendida de **utmp** (**utmpx**) con campos adicionales.

19.4.5.- *lastlog*

El archivo **lastlog** es un fichero binario guardado generalmente en **/var/adm/**, y que contiene un registro para cada usuario con la fecha y hora de su

última conexión; podemos visualizar estos datos para un usuario dado mediante la orden **finger**:

```
anita:/# finger toni
Login name: toni In real life: Toni at ANITA
Directory: /export/home/toni Shell: /bin/sh
Last login Mon Mar 6 11:07 on pts/11 from localhost
No unread mail
No Plan.
anita:/#
```

19.4.6.- *faillog*

Este fichero es equivalente al anterior, pero en lugar de guardar información sobre la fecha y hora del último acceso al sistema lo hace del último intento de acceso de cada usuario; una conexión es fallida si el usuario (o alguien en su lugar) teclea incorrectamente su contraseña. Esta información se muestra la siguiente vez que dicho usuario entra correctamente a la máquina:

```
andercheran login: toni
Password:
Linux 2.0.33.
1 failure since last login. Last was 14:39:41 on tty9.
Last login: Wed May 13 14:37:46 on tty9 from pleione.cc.upv.es.
andercheran:~$
```

19.4.7.- *loginlog*

Si en algunas versiones de Unix (como Solaris) creamos el archivo **/var/adm/loginlog** (que originalmente no existe), se registrarán en él los intentos fallidos de **login**, siempre y cuando se produzcan cinco o más de ellos seguidos:

```
anita:/# cat /var/adm/loginlog
toni:/dev/pts/6:Thu Jan 6 07:02:53 2000
toni:/dev/pts/6:Thu Jan 6 07:03:00 2000
toni:/dev/pts/6:Thu Jan 6 07:03:08 2000
toni:/dev/pts/6:Thu Jan 6 07:03:37 2000
toni:/dev/pts/6:Thu Jan 6 07:03:44 2000
anita:/#
```

19.4.8.- *btmpt*

En algunos clones de Unix, como Linux o HP-UX, el fichero **btmpt** se utiliza para registrar las conexiones fallidas al sistema, con un formato similar al que **wtmp** utiliza para las conexiones que han tenido éxito:

```
andercheran:~# last -f /var/adm/btmp |head -7
pnvarro ttyq1 term104.aiind.up Wed Feb 9 16:27 - 15:38 (23:11)
jomonra ttyq2 deportes.etsii.u Fri Feb 4 14:27 - 09:37 (9+19:09)
PNAVARRO ttyq4 term69.aiind.upv Wed Feb 2 12:56 - 13:09 (20+00:12)
panavarr ttyq2 term180.aiind.up Fri Jan 28 12:45 - 14:27 (7+01:42)
```

```
vbarbera tty0 daind03.etsii.up Thu Jan 27 20:17 still logged in
pangel ttyq1 agarcia2.ter.upv Thu Jan 27 18:51 - 16:27 (12+21:36)
abarra tty0 dtra-51.ter.upv. Thu Jan 27 18:42 - 20:17 (01:34)
andercheran:~#
```

19.4.9.- *su*log

Este es un fichero de texto donde se registran las ejecuciones de la orden **SU**, indicando fecha, hora, usuario que lanza el programa y usuario cuya identidad adopta, terminal asociada y éxito (^+) o fracaso (^-) de la operación:

```
anita:/# head -4 /var/adm/sulog
SU 12/27 07:41 + console root-toni
SU 12/28 23:42 - vt01 toni-root
SU 12/28 23:43 + vt01 toni-root
SU 12/29 01:09 + vt04 toni-root
anita:/#
```

19.4.10.- *debug*

En este archivo de texto se registra información de depuración (de *debug*) de los programas que se ejecutan en la máquina; esta información puede ser enviada por las propias aplicaciones o por el núcleo del sistema operativo:

```
luisa:~# tail -8 /var/adm/debug
Dec 17 18:51:50 luisa kernel: ISO9660 Extensions: RRIP_1991A
Dec 18 08:15:32 luisa sshd[3951]: debug: sshd version 1.2.21
[i486-unknown-linux]
Dec 18 08:15:32 luisa sshd[3951]: debug: Initializing random number
generator; seed file /etc/ssh_random_seed
Dec 18 08:15:32 luisa sshd[3951]: debug: inetd sockets after
dupping: 7, 8
Dec 18 08:15:34 luisa sshd[3951]: debug: Client protocol version
1.5; client
software version 1.2.21
Dec 18 08:15:34 luisa sshd[3951]: debug: Calling cleanup
0x800cf90(0x0)
Dec 18 16:33:59 luisa kernel: VFS: Disk change detected on device
02:00
Dec 18 23:41:12 luisa identd[2268]: Successful lookup: 1593 , 22 :
toni.users
luisa:~#
```

19.5.- LOGS REMOTOS

El demonio **syslog** permite fácilmente guardar registros en máquinas remotas; de esta forma se pretende que, aunque la seguridad de un sistema se vea comprometida y sus *logs* sean modificados se puedan seguir registrando las actividades sospechosas en una máquina *a priori* segura. Esto se consigue definiendo un **'LOGHOST'** en lugar de un archivo normal en el fichero **/etc/syslogd.conf** de la máquina de la que nos interesa guardar información; por

ejemplo, si queremos registrar toda la información de prioridad **info** y **notice** en la máquina remota **rosita**, lo indicaremos de la siguiente forma:

```
*.=info;*.=notice @rosita
```

Tras modificar **/etc/syslogd.conf** hacemos que el demonio relea su fichero de configuración enviándole la señal **sighup** (por ejemplo, con **kill**). Por su parte, en el **host** donde deseemos almacenar los **logs**, tenemos que tener definido el puerto **syslog** en **/etc/services** y ejecutar **syslogd** con el parámetro **-r** para que acepte conexiones a través de la red:

```
rosita:~# grep syslog /etc/services
syslog 514/udp
rosita:~# ps xua|grep syslogd
root 41 0.0 0.4 852 304 ? S Mar21 0:01 /usr/sbin/syslogd
rosita:~# kill -TERM 41
rosita:~# syslogd -r
rosita:~#
```

A partir de ese momento todos los mensajes generados en la máquina origen se enviarán a la destino y se registrarán según las reglas de ésta, en un fichero (lo habitual), en un dispositivo. . . o incluso se reenviarán a otra máquina (en este caso hemos de tener cuidado con los bucles); si suponemos que estas reglas, en nuestro caso, registran los mensajes de la prioridad especificada antes en **/var/adm/messages**, en este archivo aparecerán entradas de la máquina que ha enviado la información:

```
rosita:~# tail -3 /var/adm/messages
Mar 23 07:43:37 luisa syslogd 1.3-3: restart.
Mar 23 07:43:46 luisa in.telnetd[7509]: connect from amparo
Mar 23 07:57:44 luisa -- MARK --
rosita:~#
```

Esto, que en muchas situaciones es muy recomendable, si no se realiza correctamente puede incluso comprometer la seguridad de la máquina que guarda registros en otro equipo: por defecto, el tráfico se realiza en texto claro, por lo que cualquier atacante con un **sniffer** entre las dos máquinas puede tener acceso a información importante que habría que mantener en secreto; imaginemos una situación muy habitual: un usuario que teclea su **password** cuando el sistema le pide el **login**.

Evidentemente, esto generará un mensaje de error que **syslogd** registrará; este mensaje será similar a este (Linux Slackware 4):

```
Mar 23 05:56:56 luisa login[6997]: invalid password for `UNKNOWN'\
on `tty5' from `amparo'
```

Pero, ¿qué sucedería si en lugar de **`UNKNOWN'** el sistema almacenara el nombre de usuario que se ha introducido, algo que hacen muchos clones de Unix? En esta situación el mensaje sería muy parecido al siguiente (Linux Red Hat 6.1):

```
Mar 23 05:59:15 rosita login[3582]: FAILED LOGIN 1 FROM amparo FOR\
5k4@b&-, User not known to the underlying authentication module
```

Como podemos ver se registraría una contraseña de usuario, contraseña que estamos enviando a la máquina remota en texto claro a través de la red; evidentemente, es un riesgo que no podemos correr. Quizás alguien pueda pensar que una clave por sí sola no representa mucho peligro, ya que el atacante no conoce el nombre de usuario en el sistema. De ninguna forma: el pirata sólo tiene que esperar unos instantes, porque cuando el usuario teclee su *login* y su *password* correctamente (en principio, esto sucederá poco después de equivocarse, recordemos que el usuario trata de acceder a su cuenta) el sistema generará un mensaje indicando que ese usuario (con su nombre) ha entrado al sistema.

Para evitar este problema existen dos aproximaciones: o bien registramos *logs* en un equipo directamente conectado al nuestro, sin emitir tráfico al resto de la red, o bien utilizamos comunicaciones cifradas (por ejemplo con *ssh*) para enviar los registros a otro ordenador. En el primer caso sólo necesitamos un equipo con dos tarjetas de red, una por donde enviar el tráfico hacia la red local y la otra para conectar con la máquina donde almacenamos los *logs*, que sólo será accesible desde nuestro equipo y que no ha de tener usuarios ni ofrecer servicios; no es necesaria una gran potencia de cálculo: podemos aprovechar un viejo 386 o 486 con Linux o FreeBSD para esta tarea.

El segundo caso, utilizar comunicaciones cifradas para guardar registros en otro equipo de la red, requiere algo más de trabajo; aquí no es estrictamente necesario que la máquina esté aislada del resto de la red, ya que la transferencia de información se va a realizar de forma cifrada, consiguiendo que un potencial atacante no obtenga ningún dato comprometedor analizando el tráfico; evidentemente, aunque no esté aislado, es fundamental que el sistema donde almacenamos los *logs* sea seguro. Para enviar un *log* cifrado a una máquina remota podemos utilizar, como hemos dicho antes, *ssh* unido a las facilidades que ofrece *syslogd*; si lo hacemos así, lo único que necesitamos es el servidor *sshd* en la máquina destino y el cliente *ssh* en la origen. Por ejemplo, imaginemos que queremos utilizar a *rosita* para almacenar una copia de los registros generados en *luisa* conforme se vayan produciendo; en este caso vamos a enviar *logs* a un *fifo* con nombre, desde donde los cifraremos con *ssh* y los enviaremos al sistema remoto a través de la red. Lo primero que necesitamos hacer es crear un fichero de tipo tubería en la máquina origen, por ejemplo con *mknod* o *mkfifo*:

```
luisa:~# mknod /var/run/cifra p
luisa:~# chmod 0 /var/run/cifra
luisa:~# ls -l /var/run/cifra
p----- 1 root root 0 May 4 05:18 /var/run/cifra
luisa:~#
Este es el archivo al que enviaremos desde syslogd los registros que
nos interesen, por ejemplo los de
prioridad warn; hemos de modificar /etc/syslog.conf para añadirle
una línea como la siguiente:
luisa:~# tail -1 /etc/syslog.conf
*.warn | /var/run/cifra
luisa:~#
```


A continuación haremos que **syslog** relea su nueva configuración mediante la señal **sighup**:

```
luisa:~# ps xua|grep syslog |grep -v grep
root 7978 0.0 0.2 1372 156 ? S 03:01 0:00 syslogd -m 0
luisa:~# kill -HUP 7978
luisa:~#
```

Una vez realizados estos pasos ya conseguimos que se registren los eventos que nos interesan en el fichero **/var/run/cifra**; este archivo es una tubería con nombre, de forma que los datos que le enviamos no se graban en el disco realmente, sino que sólo esperan a que un proceso lector los recoja. Ese proceso lector será justamente el cliente **ssh**, encargado de cifrarlos y enviarlos al sistema remoto; para ello debemos lanzar una orden como:

```
luisa:~# cat /var/run/cifra | ssh -x rosita 'cat >>/var/log/luisa'
```

Si tenemos configurado **ssh** para que autentique sin clave podemos lanzar el proceso directamente en **background**; si tenemos que introducir la clave del **root**, una vez tecleada podemos parar el proceso y relanzarlo también en segundo plano (esto es simplemente por comodidad, realmente no es necesario). Lo único que estamos haciendo con este mecanismo es cifrar lo que llega al **fifo** y enviarlo de esta forma al sistema remoto, en el que se descifrá y se guardará en el fichero **/var/log/luisa**.

Quizás nos interese añadir unas líneas en los **scripts** de arranque de nuestra máquina para que este proceso se lance automáticamente al iniciar el sistema; si lo hacemos así hemos de tener cuidado con la autenticación, ya que si **ssh** requiere una clave para conectar con el sistema remoto es probable que la máquina tarde más de lo normal en arrancar si un operador no está en la consola: justamente el tiempo necesario hasta que **ssh** produzca un **timeout** por no teclear el **password** de **root** en el sistema remoto. Si al producirse el **timeout** el programa **ssh** no devuelve el control al **shell**, el sistema ni siquiera arrancará; de cualquier forma, si ese **timeout** se produce **no** estaremos registrando ningún evento en la otra máquina. Por supuesto, también debemos prestar atención a otros problemas con la máquina destino que eviten que la conexión se produzca, con un número máximo de usuarios sobrepasado o simplemente que ese sistema esté apagado.

El sistema de red

Tan importante como el perfecto manejo del sistema de ficheros o la asignación de permisos es el buen funcionamiento de la red basada en UNIX. Para ello deberemos conocer los ficheros más importantes así como las órdenes necesarias para administrar correctamente la red.

20.1.- ALGUNOS FICHEROS IMPORTANTES

En esta sección se abordarán sólo algunos ficheros que no han sido tratados en la parte dedicada a la administración de redes. Para alguna duda sobre ficheros, recomendamos dirigirse a dicha sección.

20.1.1.- /etc/ethers

De la misma forma que en /etc/hosts se establecía una correspondencia entre nombres de máquina y sus direcciones **ip**, en este fichero se establece una correspondencia entre nombres de máquina y direcciones **ethernet**, en un formato muy similar al archivo /etc/hosts:

```
00:20:18:72:c7:95 pleione.cc.upv.es
```

En la actualidad el archivo /etc/ethers no se suele encontrar (aunque para el sistema sigue conservando su funcionalidad, es decir, si existe se tiene en cuenta) en casi ninguna máquina Unix, ya que las direcciones hardware se obtienen por **arp**.

20.1.2.- /etc/hosts.equiv

En este fichero se indican, una en cada línea, las máquinas confiables. ¿Qué significa **confiables**? Básicamente que confiamos en su seguridad tanto como en la nuestra, por lo que para facilitar la compartición de recursos, no se van a pedir contraseñas a los usuarios que quieran conectar desde estas máquinas con el mismo **login**, utilizando las órdenes **bsd r*** (**rlogin**, **rsh**, **rcp**. . .). Por ejemplo, si en el fichero /etc/hosts.equiv del servidor **maquina1** hay una entrada para el nombre de **host** **maquina2**, cualquier usuario de este sistema puede ejecutar una orden como la siguiente para conectar a **maquina1** ¡sin necesidad de ninguna clave!:

```

maquina2:~$ rlogin maquina1
Last login: Sun Oct 31 08:27:54 from localhost
Sun Microsystems Inc. SunOS 5.7 Generic October 1998
maquina1:~$

```

Obviamente, esto supone un gran problema de seguridad, por lo que lo más recomendable es que el fichero `/etc/hosts.equiv` esté vacío o no exista. De la misma forma, los usuarios pueden crear ficheros `$HOME/.rhosts` para establecer un mecanismo de confiabilidad bastante similar al de `/etc/hosts.equiv`; es importante para la seguridad de nuestro sistema el controlar la existencia y el contenido de estos archivos `.rhosts`. Por ejemplo, podemos aprovechar las facilidades de planificación de tareas de Unix para, cada cierto tiempo, chequear los directorios `$HOME` de los usuarios en busca de estos ficheros, eliminándolos si los encontramos. Un *shellscript* que hace esto puede ser el siguiente:

```

#!/bin/sh
for i in `cat /etc/passwd |awk -F: '{print $6}'`; do
cd $i
if [ -f .rhosts ]; then
echo "$i/.rhosts detectado"|mail -s "rhosts" root
rm -f $i/.rhosts
fi
done

```

Este programa envía un correo al **root** en caso de encontrar un fichero `.rhosts`, y lo elimina; podemos planificarlo mediante **cron** para que se ejecute, por ejemplo, cada cinco minutos (recomendamos volver a la sección donde se explicó el comando **crontab**).

20.1.3.- El fichero `.netrc`

El mecanismo de autenticación que acabamos de ver sólo funciona con los **ordenes r*** de Unix **bsd**; la conexión vía **ftp** seguirá solicitando un nombre de usuario y una clave para acceder a sistemas remotos. No obstante, existe una forma de automatizar **ftp** para que no solicite estos datos, y es mediante el uso de un archivo situado en el directorio `$HOME` de cada usuario (en la máquina desde donde se invoca a **ftp**, no en la servidora) y llamado `.netrc`. En este fichero se pueden especificar, en texto claro, nombres de máquina, nombres de usuario y contraseñas de sistemas remotos, de forma que al conectar a ellos la transferencia de estos datos se realiza automáticamente, sin ninguna interacción con el usuario. Por ejemplo, imaginemos que el usuario **root** del sistema **luisa** conecta habitualmente a **rosita** por **ftp**, con nombre de usuario ``toni'`; en su `$HOME` de **luisa** puede crear un fichero `.netrc` como el siguiente:

```

luisa:~# cat $HOME/.netrc
machine rosita
login toni
password h/I0&54
luisa:~#

```

Si este archivo existe, cuando conecte al sistema remoto no se le solicitará ningún nombre de usuario ni contraseña:

```
luisa:~# ftp rosita
Connected to rosita.
220 rosita FTP server (Version wu-2.6.0(1) Thu Oct 21 12:27:00 EDT 1999) ready.
331 Password required for toni.
230 User toni logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

La existencia de ficheros **.netrc** en los **\$HOME** de los usuarios se puede convertir en un grave problema de seguridad: si un atacante consigue leer nuestro fichero **.netrc**, automáticamente obtiene nuestro nombre de usuario y nuestra clave en un sistema remoto. Por tanto, no es de extrañar que para que el mecanismo funcione correctamente, este fichero sólo puede ser leído por su propietario; si no es así, no se permitirá el acceso al sistema remoto (aunque los datos de **.netrc** sean correctos):

```
luisa:~# ftp rosita
Connected to rosita.
220 rosita FTP server (Version wu-2.6.0(1) Thu Oct 21 12:27:00 EDT
1999) ready.
Error - .netrc file not correct permissions.
Remove password or correct mode (should be 600).
ftp>
```

Existe una diferencia abismal entre el uso de **.rhosts** y el de **.netrc**; en el primer caso al menos conseguimos que nuestra clave no se envíe a través de la red, pero mediante **.netrc** lo único que conseguimos es no tener que teclear la clave y el **login** explícitamente: se envían de forma automática.

Además de esto, si alguien consigue privilegios de administrador en la máquina cliente, podrá leer los posibles archivos **.netrc** que sus usuarios posean; por tanto, este mecanismo sólo se ha de utilizar para conexiones a sistemas remotos como usuario anónimo (**anonymous** o **ftp**). Quizás nos convenga rastrear periódicamente los directorios de conexión de nuestros usuarios en busca de archivos **.netrc**, por ejemplo mediante un **shellscript** muy similar al que hemos visto para buscar ficheros **.rhosts**.

20.2.- SERVICIOS BÁSICOS DE RED

Dentro de este apartado vamos a comentar brevemente la función de algunos servicios de Unix y sus potenciales problemas de seguridad. Los aquí expuestos son servicios que habitualmente han de estar **cerrados**, por lo que no implican excesivos problemas de seguridad conocidos. Así, no vamos a entrar en muchos detalles con ellos.

20.2.1.- *systat*

El servicio **systat** se asocia al puerto 11 de una máquina Unix, de forma que al recibir una petición mediante **tcp** el demonio **inetd** ofrece una imagen de la tabla de procesos del sistema, por ejemplo ejecutando una orden como **ps -auwwx** en Linux o **ps -ef** en Solaris; en algunos Unices se ofrece la salida de órdenes como **who** o **w** en lugar de la tabla de procesos: es fácil configurar lo que cada administrador desee mostrar simplemente modificando la línea correspondiente de **/etc/inetd.conf**:

```
anita:~# grep systat /etc/inetd.conf
systat stream tcp nowait root /usr/bin/ps ps -ef
anita:~#
```

Bien se ofrezca la tabla de procesos o bien otro tipo de información sobre el sistema, este servicio es habitual encontrarlo **deshabilitado**, ya que cualquier dato sobre nuestro sistema (especialmente procesos, nombres de usuario, máquinas desde las que conectan. . .) puede ser aprovechado por un pirata para atacar el equipo. Si por motivos de comodidad a la hora de administrar varios **hosts** dentro de una red local necesitamos tener abierto **systat**, debemos restringir las direcciones desde las que se puede acceder al servicio mediante **TCP Wrappers**.

20.2.2.- *daytime*

El servicio **daytime**, asociado al puerto 13, tanto **tcp** como **udp**, es un servicio interno de **inetd** (esto es, no hay un programa externo que lo sirva, el propio **inetd** se encarga de ello); al recibir una conexión a este puerto, el sistema mostrará la fecha y la hora, en un formato muy similar al resultado de la orden **date**:

```
anita:~# telnet rosita daytime
Trying 195.195.5.1...
Connected to rosita.
Escape character is '^]'.
Thu Apr 20 05:02:33 2000
Connection closed by foreign host.
anita:~#
```

Aunque a primera vista este servicio no represente un peligro para la integridad de nuestro sistema, siempre hemos de recordar una norma de seguridad fundamental: sólo hay que ofrecer los servicios estrictamente necesarios para el correcto funcionamiento de nuestras máquinas. Como **daytime** no es un servicio básico, suele ser recomendable cerrarlo; además, la información que proporciona, aunque escasa, puede ser suficiente para un atacante: le estamos indicando el estado del reloj de nuestro sistema, lo que por ejemplo le da una idea de la ubicación geográfica del equipo.

Un servicio parecido en muchos aspectos a **daytime** es **time** (puerto 37, **tcp** y **udp**); también indica la fecha y hora del equipo, pero esta vez en un formato que no es inteligible para las personas:

```

anita:~# telnet rosita time
Trying 195.195.5.1...
Connected to rosita.
Escape character is '^]'.
['^Connection closed by foreign host.
anita:~#

```

Este servicio suele ser más útil que el anterior: aunque una persona no entienda la información mostrada por *time*, sí que lo hace una máquina Unix. De esta forma, se utiliza *time* en un servidor para que las estaciones cliente puedan sincronizar sus relojes con él con órdenes como *netdate* o *rdate*:

```

luisa:~# date
Thu Apr 20 02:19:15 CEST 2000
luisa:~# rdate rosita
[rosita] Thu Apr 20 05:10:49 2000
luisa:~# date
Thu Apr 20 02:20:02 CEST 2000
luisa:~# rdate -s rosita
luisa:~# date
Thu Apr 20 05:11:59 2000
luisa:~#

```

Los problemas de *time* son en principio los mismos que los de *daytime*; aunque también es recomendable mantener este servicio cerrado, es más fácil imaginar situaciones en las que un administrador desee ofrecer *time* en varias máquinas que imaginar la necesidad de ofrecer *daytime*.

20.2.3.- *chargen*

chargen (puerto 19, *tcp* y *udp*) es un generador de caracteres servido internamente por *inetd*, que se utiliza sobre todo para comprobar el estado de las conexiones en la red; cuando alguien accede a este servicio simplemente ve en su terminal una secuencia de caracteres ASCII que se repite indefinidamente.

Los posibles problemas de seguridad relacionados con *chargen* suelen ser negaciones de servicio, tanto para la parte cliente como para la servidora. Sin duda el ejemplo más famoso de utilización de *chargen* es una de las anécdotas del experto en seguridad Tsutomu Shimomura (el principal contribuidor en la captura de Kevin Mitnick, el pirata más famoso de los noventa): cuando conectaba a un servidor de *ftp* anónimo, Shimomura se dió cuenta de que la máquina lanzaba un *finger* contra el cliente que realizaba la conexión. Esto no le gustó, y decidió comprobar si ese sistema utilizaba el *finger* habitual; para ello modificó el fichero */etc/inetd.conf* de su sistema de forma que las peticiones *finger* se redirigieran al generador de caracteres *chargen*. Conectó al servidor de nuevo, y al hacer éste otro *finger*, la máquina de Shimomura se dedicó a enviar *megas* y *megas* de caracteres (*chargen* no finaliza hasta que el cliente corta la conexión); en unas pocas horas el sistema remoto quedó inoperativo, y a la mañana siguiente ese *finger* automático había sido eliminado de la configuración del servidor. Ese servidor no habría sufrido una caída si hubiera utilizado *safe finger*, un programa

de Wietse Venema que se distribuye junto a *TCP Wrappers* y que limita la potencial cantidad de información que *finger* puede recibir.

20.2.4.- *finger*

Típicamente el servicio *finger* (puerto 79, *tcp*) ha sido una de las principales fuentes de problemas de Unix. Este protocolo proporciona información – demasiado detallada – de los usuarios de una máquina, estén o no conectados en el momento de acceder al servicio; para hacerlo, se utiliza la aplicación *finger* desde un cliente, dándole como argumento un nombre de máquina precedido del símbolo '@' y, opcionalmente, de un nombre de usuario (*finger* sobre el sistema local no utiliza el servicio de red, por lo que no lo vamos a comentar aquí). En el primer caso, *finger* nos dará datos generales de los usuarios conectados en ese momento a la máquina, y en el segundo nos informará con más detalle del usuario especificado como parámetro, esté o no conectado:

```
anita:~# finger @rosita
[rosita]
Login Name Tty Idle Login Time Office Office Phone
toni Toni at ROSITA */0 28 Apr 20 04:43 (anita)
root El Spiritu Santo 1 12 Apr 11 02:10
anita:~# finger toni@rosita
[rosita]
Login: toni Name: Toni at ROSITA
Directory: /home/toni Shell: /bin/bash
On since Thu Apr 20 04:43 (CEST) on pts/0 from anita
30 minutes 28 seconds idle
(messages off)
No mail.
No Plan.
anita:~#
```

Como podemos ver, *finger* está proporcionando mucha información que podría ser de utilidad para un atacante: nombres de usuario, hábitos de conexión, cuentas inactivas. . . incluso algunas organizaciones rellenan exhaustivamente el campo *gecos* del fichero de contraseñas, con datos como números de habitación de los usuarios o incluso su teléfono. Está claro que esto es fácilmente aprovechable por un pirata para practicar ingeniería social contra nuestros usuarios – o contra el propio administrador –. Es **básico** para la integridad de nuestras máquinas **deshabilitar** este servicio, restringir su acceso a unos cuantos equipos de la red local mediante *TCP Wrappers* o utilizar versiones del demonio *fingerd* como *ph* (*Phone Book*), que permiten especificar la información que se muestra al acceder al servicio desde cada máquina.

20.2.5.- *auth*

Se llama *socket* a la combinación de una dirección de máquina y un puerto; esta entidad identifica un proceso único en la red. Un par de *sockets*, uno en la máquina receptora y otro en la emisora definen una conexión en protocolos como *tcp*; esta conexión también será única en la red en un instante dado. Como vemos, no entra en juego ningún nombre de usuario: en *tcp/ip* se establecen canales de comunicación entre máquinas, no entre personas; no obstante, en muchas

ocasiones nos puede interesar conocer el nombre de usuario bajo el que cierta conexión se inicia.

Por ejemplo, de esta forma podríamos ofrecer o denegar un servicio en función del usuario que lo solicita, aparte de la máquina desde donde viene la petición.

El protocolo *auth* (puerto 113, *tcp*) viene a solucionar este problema con un esquema muy simple: cuando un servidor necesita determinar el usuario que ha iniciado una conexión contacta con el demonio *identd* y le envía los datos necesarios para distinguir dicha conexión (los componentes de los dos *sockets* que intervienen) de las demás. De esta forma, el demonio identifica al usuario en cuestión y devuelve al servidor información sobre dicho usuario, generalmente su *login*. Por ejemplo, si utilizamos *TCP Wrappers* – un programa servidor que utiliza este mecanismo para determinar nombres de usuario siempre que sea posible –, se registrará el *login* del usuario remoto que solicita un servicio en nuestra máquina si el sistema remoto tiene habilitado *auth*:

```
luisa:~# tail -2 ~adm/syslog
Apr 24 04:16:19 luisa wu.ftpd[1306]: connect from rosita
Apr 24 04:16:21 luisa ftpd[1306]: ANONYMOUS FTP LOGIN FROM \
rosita [195.195.5.1], toni@
luisa:~#
```

No obstante, si el sistema desde el que esa persona conecta no tiene habilitado dicho servicio, el nombre de usuario no se va a poder conseguir:

```
luisa:~# tail -2 ~adm/syslog
Apr 24 04:19:37 luisa wu.ftpd[1331]: connect from root@anita
11.2. SERVICIOS B´ ASICOS DE RED 177
Apr 24 04:19:39 luisa ftpd[1331]: ANONYMOUS FTP LOGIN FROM \
root @ anita [195.195.5.3], toni@
luisa:~#
```

El servicio *auth* no se debe utilizar nunca con propósitos de autenticación obusta, ya que dependemos no de nuestros sistemas, sino de la honestidad de la máquina remota; un atacante con el suficiente nivel de privilegio en esta puede enviarnos cualquier nombre de usuario que desee. Incluso en ciertas situaciones, si *ident* no está habilitado ni siquiera hacen falta privilegios para devolver un nombre falso: cualquier usuario puede hacerlo. En cambio, sí que es útil para detectar pequeñas violaciones de seguridad, por lo que quizás interese habilitar el servicio en nuestras máquinas (aunque limitemos su uso mediante *TCP Wrappers*.)

Índice

A

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2
 - Índice 3, 3
- Índice 1, 1
- Índice 1, 1

B

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2

C

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1

D

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1

E

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1

G

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1

H

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1

K

- Índice 1, 1

L

- Índice 1, 1
 - Índice 2, 2
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1

M

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2

N

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2
- Índice 1, 1

- Índice 1, 1
- Índice 1, 1

R

- Índice 1, 1
- Índice 1, 1

S

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1

T

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2

W

- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
 - Índice 2, 2
- Índice 1, 1
- Índice 1, 1
- Índice 1, 1
- Índice ,

